Universidade de Aveiro
2015

Departamento de Eletrónica,
Telecomunicações e Informática

Sandro
Rodrigues

Infraestrutura Operacional para um sistema baseado em FPGA e ARM

Operating infrastructure for an FPGA and ARM system

Universidade de Aveiro
2015

Departamento de Eletrónica,
Telecomunicações e Informática

Sandro Rodrigues

Infraestrutura Operacional para um sistema baseado em FPGA e ARM

Operating infrastructure for an FPGA and ARM system

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica do Doutor Fritz Mayer-Lindenberg, Professor Catedrático do Instituto de Tecnologias Computacionais da Universidade Técnica de Hamburgo, do Doutor Heiko Falk, Professor Catedrático do Instituto de Sistemas Embutidos da Universidade Técnica de Hamburgo, e sob a co-orientação científica da Doutora Iouliia Skliarova, Professora auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

Prof. Doutor Fritz Mayer-Lindenberg
Professor Catedrático do Instituto de Tecnologias Computacionais da Universidade Técnica de Hamburgo(Orientador)

Prof. Doutor Heiko Falk
Professor Catedrático do Instituto de Tecnologias Computacionais da Universidade Técnica de Hamburgo

Prof. Doutora Iouliia Skliarova
Professora Auxiliar da Universidade de Aveiro(Co-Orientadora)

agradecimentos /
acknowledgements

First of all, I want to thank my supervisor, Prof. Fritz Mayer-Lindenberg, as well as Prof. Iouliia Skliarova, for the useful comments, remarks and engagement throughout the development of this thesis.
Very special thanks to Yaroslav Gevorkov, for his patience and help in the explanation of some tasks of this thesis, as well as Rui Brito for his friendship and his helpful comments.
I would like to thank my parents for the unconditional support throughout my education process, always believing in my abilities. I am especially grateful to my brother, Fabio Rodrigues, who was always there whenever I needed his help in any given topic.
I cannot thank Soraia Oliveira enough for her unconditional support, her constant presence, her helpful advices, her infinite patience, among so many others. She was the pillar that give me the necessary strength to overcome my difficulties.
Finally, I would like to express immense gratitude to all my friends across the globe, who were not mentioned but not forgotten.

Palavras Chave

FPGA, Infraestrutura, Sistemas Reconfiguráveis, Co-processadores, Software e Hardware, Sistemas Embutidos.

Resumo

Devido aos avanços na tecnologia FPGA e à crescente necessidade de maior capacidade de processamento, surgiram sistemas programáveis que incorporam um sistema de processamento e uma componente de lógica programável. Assim, possibilitando o desenvolvimento de sistemas computacionais especializados para uma ampla gama de aplicações práticas, incluindo processamento de dados e sinal, computação de alto desempenho, sistemas embutidos, entre muitos outros.

Para dar lugar a uma infraestrutura capaz de usar os benefícios de um sistema tão reconfigurável, os principais objetivos desta tese são implementar uma infraestrutura composta por hardware, software e recursos de rede, que incorpora serviços para operar e gerir a interface dos periféricos, constituido por blocos de construção básicos necessários para a execução de aplicações. O projeto será desenvolvido utilizando um chip da família Zynq-7000 do fabricante Xilinx.

Keywords

FPGA, Infrastructure, Reconfigurable Systems, Co-processors, Software and Hardware, Embedded Systems.

Abstract

Advances in FPGA technology and higher processing capabilities requirements have pushed to the emerge of All Programmable Systems-on-Chip, which incorporate a hard designed processing system and a programmable logic that enable the development of specialized computer systems for a wide range of practical applications, including data and signal processing, high performance computing, embedded systems, among many others.
To give place to an infrastructure that is capable of using the benefits of such a reconfigurable system, the main goal of the thesis is to implement an infrastructure composed of hardware, software and network resources, that incorporates the necessary services for the operation, management and interface of peripherals, that coompose the basic building blocks for the execution of applications.
The project will be developed using a chip from the Zynq-7000 All Programmable Systems-on-Chip family.

Contents

List of Figures

List of Tables

Listings

List of Acronyms

ACP	Accelerator Coherency Port
APSoC	All Programmable System on Chip
APU	Application Processor Unit
ASCII	American Standard Code for Information Interchange
ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
BRAM	Block Random Access Memory
BSP	Board Support Package
CAD	Computer-Aided Design
CLB	Configurable Logic Block
DDR	Double Data Rate
DMA	Direct Memory Access
DSP	Digital Signal Processing
ELF	Executable and Linkable Format
EMIO	Extented MIO
FF	Flip-Flop
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSBL	First Stage Boot Loader
FSM	Finite State Machine
GIC	General Interrupt Controller
GP	General Purpose
GPIO	General Purpose Input Output
GUI	Graphical User Interface

HDL	Hardware Description Language
HDMI	High-Definition Multimedia Interface
HP	High Performance
I/O	Input/Output
IC	Integrated Circuit
IOB	Input/Output Block
IP	Intellectual Property
ISE	Integrated Software Environment
ISR	Interrupt Service Routine
IVT	Interrupt Vector Table
JTAG	Joint Test Action Group
LUT	Look-Up Table
MIO	Multiplexed Input Output
OCM	On Chip Memory
OS	Operating System
PC	Personal Computer
PCB	Printed Circuit Board
PI	Programmable Interconnect
PL	Programmable Logic
PS	Processing System
QSPI	Quad Serial Peripheral Interface
RAM	Random Access Memory
ROM	Read Only Memory
SD	Secure Digital
SDK	Software Development Kit
SRAM	Static Random Access Memory
SoC	System On Chip
SSV	System SerVer
TCL	Tool Command Language
UART	Universal Asynchronous Receiver Transmitter

USB Universal Synchronous Bus

VGA Video Graphics Array

VHDL Very High Speed Integrated Circuit Hardware Description Language

XADC Xilinx Analog to Digital Converter

XMD Xilinx Microprocessor Debugger

Chapter 1

Introduction

1.1 Framework

Electronic devices have been increasing in complexity, operating at a very high speed, and providing more and more efficiency in the management of resources. Where the trend of electronic devices is to increase its efficiency and its operating speed even more. An example of such evolution on the electronic devices are the integrated circuits.

A large part of the electronic devices are created from integrated circuits, such as microprocessors, memory storage, peripheral controllers, digital to analogue converters, among others. Nevertheless, most of these devices were designed to carry out a specific function. But, to give answer to systems capable of performing multiple functions, the reconfigurable systems have emerged. More in particular, the Field Programmable Gate Arrays (FPGA) have enabled the user to design and implement different logical circuits on the same chip. Thus, providing great versatility and flexibility in comparison to Application Specific Integrated Circuits (ASIC).

Since the launch of the FPGA chips, many improvements have been adopted, its functional blocks have been subject to many changes. FPGAs include in its architecture several functional blocks like blocks as memory, clock managers, multiplier blocks, among others. Offering many solutions in the design of logical circuits and enabling a wider range of applications. Nowadays, FPGAs are integrated with a processing system in the same chip. The inclusion of a processing system enables the design of a complexer system, which accounts for two parallel systems processing data side by side.

To these Configurable systems adds the ability, of describing a digital circuit in a simplified way, using a Hardware Description Language (HDL), which is, after designed and synthesised, implemented on the FPGA.

1.2 Motivation

The FPGA industry is a growing market, despite the uncertain course of the future, forecasts indicate that it will continue to grow. As FPGAs are so versatile, they can be applied in many fields of application, to solve the most distinct problems. They can be applied in automotive to control multiple sensors in a car, used as a digital signal processor to process audio or video signal, in digital video cameras to control the image sensor, among many others. These devices not only overcome the standard ASICs in versatility, but offer unique and significant advantages when designing a digital system.

Advances in FPGA technology and higher processing capabilities requirements, have pushed to the emerge of All Programmable System on Chips (APSoC). Which combine a hard designed processing system and a programmable logic part. Combining these systems leads to a increased complexity in the joined system and for the designer. In this way, the advancements of FPGA also claimed for a better Software support. Appearing new design tools and other intellectual property, facilitating the designer and delivering a more efficient design flow followed by a larger range of support.

Systems that try to use the full capacities of Programmable System on Chips (PSoC) are emerging. An example of such a system is the experimental computer built of multiple PSoCs that was put together at the Institute for Computer Technologies (ICT). This system will be further analysed in the next chapter as it is a remarkable example of the use of such devices.

The FPGA is attached to a processing system in the same chip, allowing this way a high configuration of the system. The reprogramming of this system enables it to be adapted, to the applications that are to be executed. This system is adaptable on many fields of computation, making it a very versatile system.

To give place to an infrastructure that is capable of such reconfigurability, a system that incorporates such features has to be designed.

1.3 Goals

The main Goal of this Thesis is to create an infrastructure that supports applications to be executed on the referred experimental computer. The project will be carried out just on one PSoC, but the logic could be reused on all the other PSoCs.

To create such infrastructure, one has to define the main components that will make part of its architecture. Each of these components is a key feature of the infrastructure. As the system is Software and Hardware related, the infrastructure it self is based on both. There will be components composed of FPGA logic and some components based on software.

As for the hardware components, based on FPGA logic, they should grant access to the peripherals present on the hardware device. These peripherals can be of any kind, they just have to be populated on the device. The components that should be part of a general use infrastructure and that are to be linked to the design are:

- Host port interface to communicate with the processing system and address peripherals
- Controller to grant memory access
- Video output interface
- Access and communication with the co-processor unit

In the software part the design has to be capable of starting the execution of a application, commanded by the host computer. Configuring the FPGA and initiating the processing system. A link between the FPGA and the processing system is also part of the infrastructure, where a host interface on the FPGA takes care of incoming data and distributes it to the other components. This Host Port is also both software and hardware related, as it is the link of communication between the FPGA and the host computer. To address the need for a platform, that executes the required functions on the processor system, the following systems are to be integrated and developed:

- Pi-Nets Runtime System, that will be referenced in the next chapter.

- Design of a data transport layer to link the FPGA to the host computer.

For the design of the introduced infrastructure, the aims of the implementation were defined as:

- Construction of simple reusable blocks, and, when possible, reuse or integrate similar projects.
- Test the implementation and demonstrate its results.

1.4 Organisation

After this introductory chapter, the thesis will be organised as follows:

- Chapter 2 - In this chapter will be given a brief theoretical introduction to the components that create a digital system, mainly related to the FPGA devices and the Zynq-7000 from Xilinx. The Hardware tools that were used in the development of this Thesis will be discussed. Also a brief background to the ER-4 parallel computer will be given, that is the source of the problem that originated this Thesis. And, as last, the Pi-Nets runtime system that has served as base for some topics of this thesis will be presented.
- Chapter 3 - In this chapter will be a description of the followed design flow that will assist the design of a digital system. The tools that were used to design and develop this project will also be referenced.
- Chapter 4 - In this chapter, the architecture of the developed system will be presented. The infrastructure, that is to be implemented, will be analysed. Its functions, the multiple blocks and the different adoptable solutions will be discussed and compared.
- Chapter 5 - In this chapter will be described how the project was implemented in more detail. All the components of the infrastructure will be shown and its functionality will be explained.
- Chapter 6 - In this chapter is a description of the methods, in which the implementation was tested. The obtained results and the overall balance of the system will also be discussed.
- Chapter 7 - In the last Chapter of this work will be a conclusion of the global work that was developed. Also proposals for optimisation and suggestions for future work will be given.

Chapter 2

Reconfigurable Systems

In this Chapter will be given a overview of the technical aspects of a FPGA System on Chip (SoC), constituted by Programmable Logic (PL), a Application Processing Unit (APU) and multiple interfaces.

Advancements of the technology have led to the development of FPGA-based systems on chip. A System on Chip, or SoC, is a fully functional integrated circuit containing multiple components integrated on a single chip. The state of the art All Programmable System on Chip (APSoC), include an application processor unit attached to a FPGA block. The APSoC also integrates multiple communication ports to grant access to external hardware peripherals.

The SoC used in this project was the Zynq, from Xilinx. This chip is both powerful and extremely configurable, as it is essentially constituted by a Programmable Logic part, and an Application Processing System. Some aspects of its building blocks and also its communication interfaces will be further analysed.

A system that integrates such state of the art FPGAs is the ER-4. It integrates 50 FPGA SoCs in a complex network. This system will be further discussed in Section 2.5, as well as the key components of its architecture. The software developments and the applications that will run in such a system will also be presented.

2.1 FPGA integrating a Processing System

An FPGA chip is a very complex arrange of digital gates, which are designed to be programmed. An FPGA leaves the production with no functionality other than to be programmed on the desired field of application.

The basic building blocks that compose an FPGA are an array of configurable logic blocks, programmable interconnects and configurable input and output blocks. To these basic blocks, the FPGA can also include some other more specific blocks, like Random Access Memory (RAM) block, Digital Signal Processors (DSP) blocks, clock managers. The sate of the art APSoC even integrates a Processing System (PS) parallel to the FPGA. A simplified FPGA architecture that integrates a Processing System is represented in Figure 2.1. A brief reference to each of the main building blocks, will now be given, based on [7, 8, 9] and on Xilinx devices:

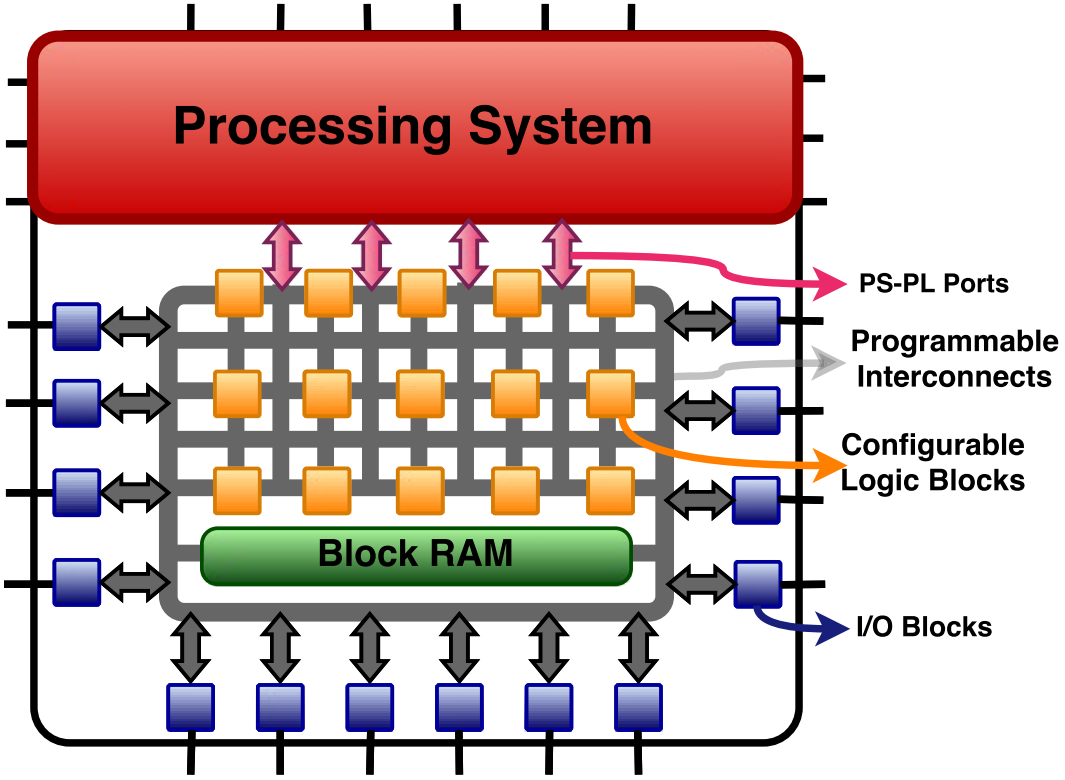

Figure 2.1: Basic architecture of a FPGA-SoC

- Configurable Logic Blocks (CLB): The logic blocks are the basic logical function elements to process the logic of the elaborated design. It can be considered the main element of an FPGA block, as it provides the storing and processing of data. The most important components to construct a Logic Block are Flip-Flops (FF) and Look-up tables (LUT). In which the LUT's contain the logical combinations produced by the design and the FF are used to store logic states or other data.
- Input and Output Blocks (IOB): An IOB interfaces the chip's external components with the logic produced within the CLBs, by buffering the data at the chip's ports. The input and output ports of the chip are fully configurable as input or output.
- Programmable Interconnect (PI): To link the configured CLBs together, and also to connect the CLBs to the IOBs, a programmable interconnect is also part of the FPGA. This interconnect links all the configured components together. The programmable interconnects are constituted by pass transistors, tri-state buffers and several multiplexers, which combined form a programmable switch matrix.

To these basic building blocks the FPGA can also contain other more complexer building blocks, such as:

- RAM blocks, also called as BRAM: Provides on-chip memory for the design. They differ in size and arrangement depending on the configuration of each FPGA architecture. They can have dual-port feature, enabling simultaneous reads and writes to two different interfaces.
- Clock Managers: The FPGA's logic is mostly synchronised to a clock signal, to process the digital signals. Special routing networks are made available to minimise skew. The

clock signals frequency rate can also be changed with help of a clock manager. The clock manager is capable of multiple features among them to enable different clock domains, with different frequencies and phase relationships.

- Digital Signal Processors (DSP): Some FPGAs integrate digital signal processing units, for a fast signal processing path. These units are very flexible as its signal pathways are very configurable. One can process multiplication or other parallel operations.

Its possible to configure a FPGA with any desired application, as its function is not restricted to any specific hardware function, it was rather build to be programmed according to the user's will. By implementing a project on an FPGA, one has the possibility to change and improve the design several times, just by reprogramming the chip. Leading to big flexibility over a hardware system that can not be changed, like Application Specific Integrated Circuits (ASIC). These systems are hard designed and can not be changed after production.

2.1.1 Integrated Processing System

Due to the increased necessity for a processing system, when designing a project for a FPGA, the FPGA chips have become so complex that they incorporate a application processing unit in its chip.

The PS can so be used as a normal processing unit, having its instruction code to execute, and can be part of a complex system combined with the FPGA, both incorporated in the same chip. Where both systems are able to process data side by side and dispose of multiple communication links between them and to the external peripherals.

Different communication links can be used by PS and PL to communicate between each other, as there were special protocols defined for this purpose. Some external hardware peripherals can also be shared between PS and PL, and they can be used as a communication link.

2.1.2 Joint Test Action Group (JTAG)

The JTAG standard was firstly introduced as a boundary scan for IC chips, to verify if the pins of the chip were correctly connected to the Printed Circuit Board (PCB), after production. As the complexity and density of PCB designs increased, the standard has also progressed to respond to the increasing needs. Mechanisms are being implemented on the IC chip to assist the JTAG connection in tests and other verification.

The JTAG works in a way that it performs a Boundary-scan, where it will detect special cells embedded in the Integrated Circuit (IC) chip device. These cells can capture input data or also force data through the JTAG access. The data is serially shifted along the scan chain, allowing that each of the boundary cells can be individually accessed. Depending on the application, the forced data can then be compared to see if the chip is behaving like it is supposed to. Refer to [10] for a more comprehensive understanding.

The standard also accounts for device specific instructions that can be used to interact with additional hardware capabilities, like accessing a microprocessor device or monitor the program execution, among many other.

The JTAG connection is very helpful when debugging the FPGA, as one can verify the functionality of some of the logic, and so be able to detect mistakes. It can also be used to program the FPGA or to program the flash memory. It's also possible to debug the APU using the JTAG interface.

2.2 Advanced eXtensible Interface (AXI)

The AXI protocol is a communication interface targeted to interconnect interfaces at high performance and high frequency. This protocol has been adopted in several FPGA based SoCs and in the communication between PS and PL.

The protocol defines as Master the interface who controls the operation of the transaction and as slave the interface that receives the order from the Master and processes it. The data can also be transferred in both directions but just one of them is master an just one is slave.

The AXI protocol is burst based, for high throughput transfers. But it defines different variants of the protocol for different types of interactions. Each interface type is defined with its own specific transfer channels and corresponding signals. An overview of the different interfaces is given bellow, for more details refer to [11].

- AXI4: The AXI4 is a burst capable transfer interface, who consists of five different channels, each of them having its own signals. The five channels can be used simultaneously. The burst mode can be configured with multiple burst options. This protocol type is the best choice for high throughput memory access.
- AXI4-Lite: The AXI-Lite interface is a smaller version of the AXI4, which is not burst capable, but covers the same transaction channels as the AXI4 interface. Using this AXI interface, one can transfer data using just the essential protocol signals. It has the lowest throughput, as it just allows one data transfer per transaction.
- AXI4-Stream: The AXI-Strem interface is defined with only one transmission channel for streaming data, where unlike the AXI4, this interface can burst unlimited amount of data. This interface has special control signals. It is designed for high throughput when using a video interface.

The architecture of the AXI4 interface and its basic signals will be further presented in the next sections, based on [1, 11].

2.2.1 AXI4 Channel Architecture

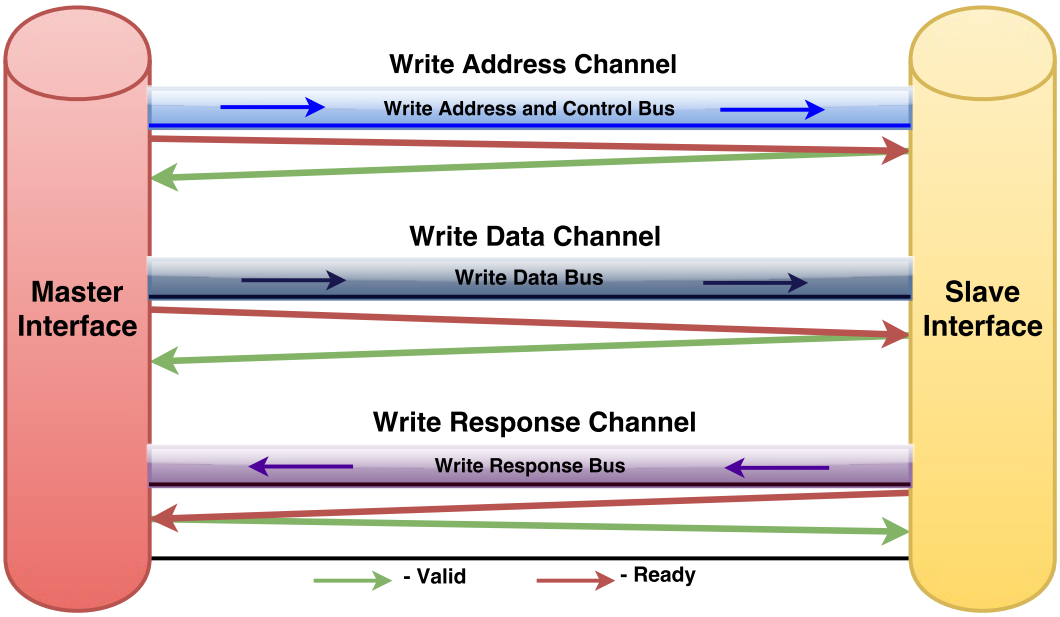

Figure 2.2: Write Channel Architecture, adapted from [1]

The three channels represented in Figure 2.2 define the AXI4 interface for a write transaction. The data, address and response channels have separate connection. The red and green arrow demonstrate the handshake mechanism. The channels are ordered from the top to bottom, being the address and control signals the first to be sent.

The Master sets the *Valid* signal when valid data and control information is accessible on the channel. The slave accepts the received data, by setting the *Ready* signal, and so acknowledging its reception to the Master.

This handshake mechanism is always used when a transaction of any of the channels occurs. The *Valid* and *Ready* signals can be used by both interface sources, depending on the direction of the data flow.

The Write transaction incorporates one extra channel to allow the slave to respond to the master, by signalising if the Write transaction has been completed correctly.

The Read channels are represented in Figure 2.3. The Read operation has small changes to the Write transaction, only two aspects differ. The direction of the handshake signals when receiving data are inverted and the response channel does not exist on the Read transaction.

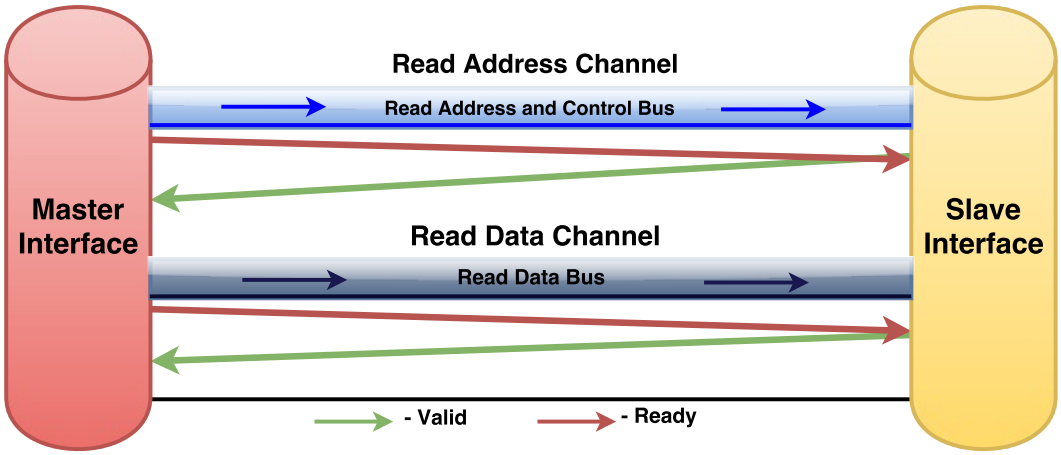

Figure 2.3: Read Channel Architecture, adapted from [1]

2.2.2 AXI4 Interconnection

The AXI protocol enables an out of order transaction scheme, by defining to each interface port an unique IDentity number (ID). With these feature one can interconnect multiple AXI interfaces and process out of order transactions. It must be assured that the order is maintained for each individual interface ID, but as for the different interfaces, transactions are tagged with its ID, enabling almost simultaneous transactions between multiple interfaces[1].

Different ID signals are defined for each channel, enabling reordering of the channel transaction of each channel, in the case of multiple interfaces in the system.

The AXI interconnect hardware Intellectual Property (IP) implements that functionality. Enabling to interconnect multiple masters to multiple slave interfaces. With this IP the user must not care about the ID's signals, as the IP takes care of that, and also deals with the mapping of the addresses from the master to the right slave interface[12].

2.3 Zynq-7000 AP SoC

The Zynq-7000 All Programmable (AP) SoC family from Xilinx is a state of the Art FPGA family, that contains besides the FPGA, an APU, that is by it self, considerably powerful. The low-end Zynq-7000 is built with a Artix-7 similar FPGA architecture and as for the APU, it contains an ARM dual-core A9. A full overview of the Zynq-7000 AP SoC is found at [13].

In Figure 2.4 its possible to see the interactions between the fundamental blocks, with its own communication links, as the configurable external IOs that are linked to the Zynq's ports.

The fundamental blocks are the FPGA part, that will be referred as PL, and the APU and all its nearby components, that will be referred as PS.

This complex system counts with multiple interfaces that are able to link both the blocks together. Some of these interfaces will be further analysed in more detail.

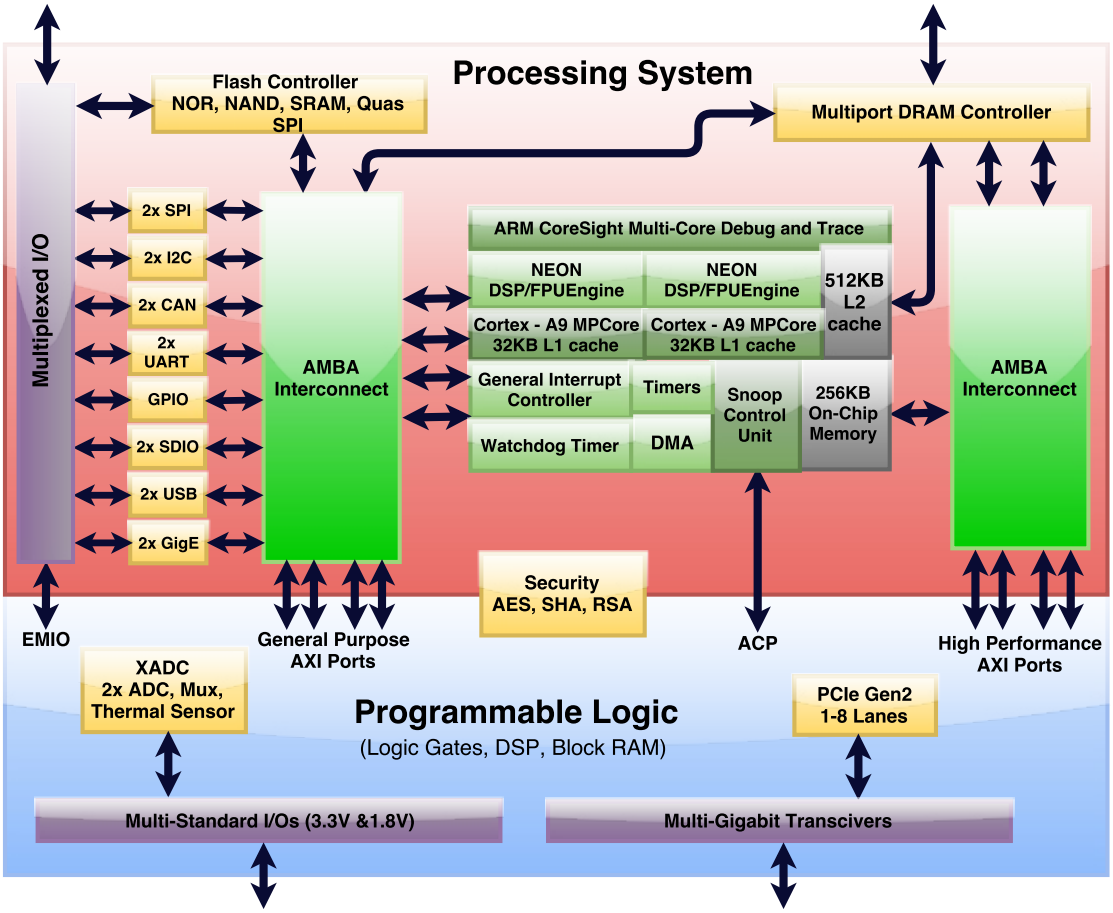

Figure 2.4: Zynq diagram, adapted from [2]

2.3.1 Processing System (PS)

The architecture of the Zynq contains an Application Processing System, which is located within the PS. The APU contains two ARM Cortex A9 processors and a respective NEON co-processor. They are connected in an multiprocessor configuration, which share 512KB of L2 cache, that can execute instructions and store data. Each ARM processor also has at its disposal a 32KB L1 cache that is capable of executing instructions or to store data. The cortex-A9 executes instructions in a 32bit, 16bit and 32bit thumb mode.

The ARM cores have a operating clock frequency of up to 999MHz, depending on the model and speed grade. The low-end Zynq-7000 with the slowest speed grade has a operating frequency of 667MHz.

A Snoop Control Unit (SCU) was integrated to the PS, to maintain the coherency of the L1 cache between both processors and the Accelerator Coherency Port (ACP) from the PL.

In parallel to the L2 cache, a On-Chip Memory (OCM) is also integrated on the PS. This OCM is 256Kb width and provides low latency access.

The PS also contains a Direct Memory Access (DMA) controller that is configured by the APU when a transfer is to be processed to the Double Data Rate (DDR) memory.

To access multiple external interfaces, the PS integrates a Multiplexed Input Output (MIO) interface, which is designed to be configured with the desired external interfaces, to the desired pin banks. This MIO also includes an option to connect the interface to peripherals present on the PL, which is called the extended MIO.

More details about the PS on the Zynq can be found on [14].

2.3.2 Programable Logic (PL)

The Zynq-7000 low-end devices contain a PL that is equipped with a Artix-7 FPGA equivalent. Which is well equipped with the following features:

- Up to 53,200 Look-Up Tables.
- Up to 106,400 Flip-Flops.
- Up to a total of 4.9Mbit in Block RAMs, with each block having 36Kbit.
- Dual 12-bit Xilinx Analog to Digital Converter (XADC), with up to 17 Differential Inputs
- Up to 220 Programmable Digital Signal Processing (DSP) slices
- Numerous configurable PS interfaces, including the Advanced eXtensible Interface (AXI) interconnects, the Extensible MIO (EMIO), the DMA controller, PL to PS interrupts and the ACP port.

For a complete description of the PL contained in the Zynq, please consult [14].

2.3.3 Processing System to Programmable Logic Interface Ports

As referenced the PL has access to multiple ports that allow it to communicate with the PS. To communicate through these interfaces the AXI protocol is used.

The Zynq has a total of four different communication interfaces between PL and PS. All the interfaces have a different application purpose and may be assigned to multiple ports. They were designed to assist the interaction within PL and PS. The available interfaces are the following:

- Four General Purpose ports (GP), in which two of them can be configured as Masters and the other two can be configured as Slaves. The four ports are 32bit width for address and for data. These ports are optimised for general use, where the PL can behave as a peripheral to the PS, or vise versa.
- Four High Performance ports (HP), that can just be configured as Masters, but with 32 or 64bit data width. These ports serve to access the external DDR memory, among other. The HP ports of the Zynq are optimised for the highest possible throughput.
- One Accelerator Coherency Port (ACP), that allows a coherent access from the PL to the PS's L2 cache and also to its OCM.
- Extended Multiplexed Input Output (EMIO), that allows access from the PS to peripherals connected to the PL.

2.3.4 Boot sources

The Zynq can be booted from multiple hardware sources, as stated by [15]. A factory flashed boot Reed Only Memory (ROM) is responsible to identify the selected boot source, by reading some mode registers. The mode registers can be changed, and so change the boot source, this process is not software assisted and must be done externally, with the help of wire jumpers or others.

The possible boot sources that can be set are Quad-Serial Peripheral Interface (QSPI), JTAG, OCM and SD card. The Zynq will boot automatically from one of the external boot sources, after each system reset. As the internal boot ROM reads the configured boot source, the first stage boot loader, contained in the source, is loaded in to memory and afterwards executed for subsequent configuration steps. The First Stage Boot Loader (FSBL) is responsible to hand-off the execution of its code to the application that is to be performed. By the end of the FSBL's execution, the FSBL will load the application to memory and start to execute the transferred application.

2.4 Development Hardware

To develop this thesis, two hardware boards containing a Zynq chip were available. Both can be used in different stages of the project, as both contain a similar Zynq chip. They have some advantages and disadvantages over the other.

The suggested development hardware for this project, was the Parallella board, as it contains the Epiphany chip, which is a powerful Floating Point Unit (FPU) with 16 processing cores. An objective of this thesis is to implement a project that is able to integrate and to make use of the capacities of the Epiphany chip. But considering the Parallella board, it does not deliver a convenient FPGA development environment, as its system is ready to boot a Linux image. All configurations are already programmed and they can not be changed, unless a JTAG cable is connected. The JTAG pins are accessible on the board, but not populated, to make them available some soldering job has to be done. Also the interfaces that the Parallella board delivers are very few considering the ZedBoard.

In the other hand the ZedBoard offers a complete development environment, where different boot options can be configured. There are many debugging tools available, like Light Emitting Diodes (LED) and switches. Also many other interfaces are available on the ZedBoard. An other advantage is that, the ZedBoard is very well documented, with many available tutorials to follow. Considering that designing a system that contains both a PS and PL, can be more complicated than just a FPGA chip, all the peripherals that the ZedBoard delivers can be extremely helpful when debugging. To facilitate the start of the project, it was chosen to first implement the project on the ZedBoard, and at the final stage when the Epiphany chip is to be implemented one would port the system to the Parallella board and continue the implementation.

The main specifications of both Hardware boards will now be presented.

2.4.1 Avnet: ZedBoard

The main specifications of the ZedBoard are as follows, for a complete reference consult [16]:

- Zynq-Z7020 with Dual-core ARM A9 CPU
- 512Mb DDR3 RAM
- 256 Mb Quad-SPI Flash

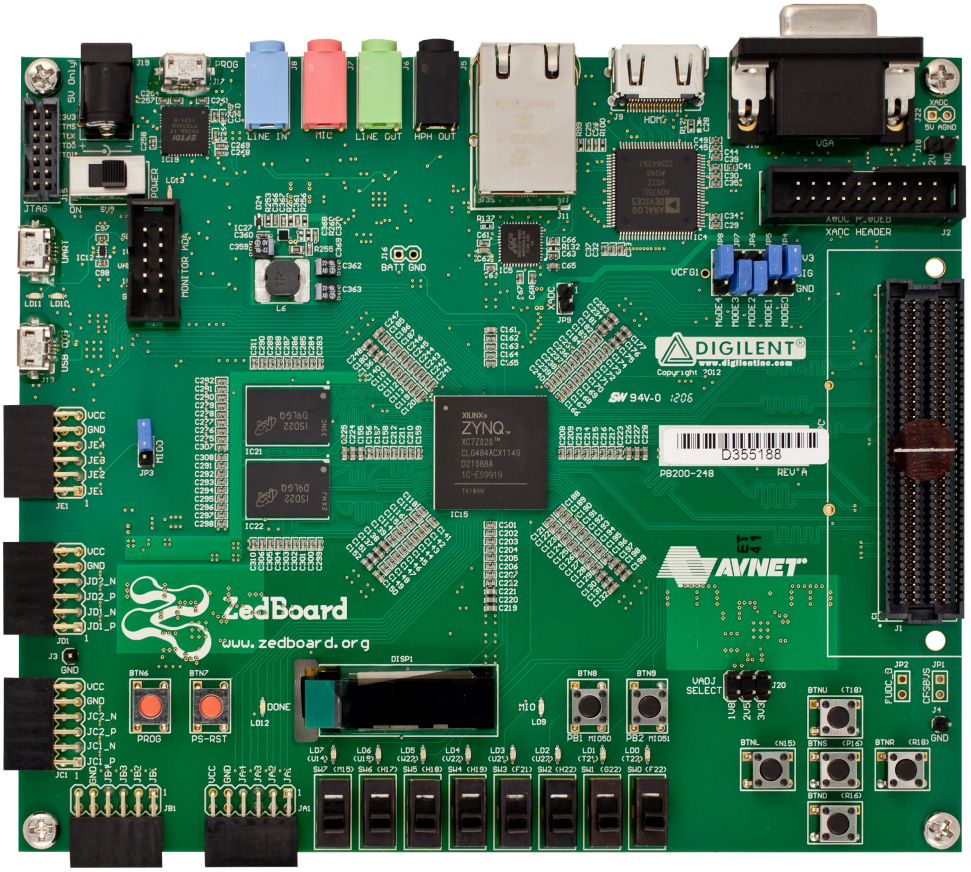

Figure 2.5: ZedBoard, from [3]

- Full size SD/MMC card cage
- Gigabit Ethernet
- HDMI output (1080p60 + audio)
- VGA connector
- 128 x 32 OLED
- 9 x User LEDs
- USB OTG and USB UART
- 5 x Pmod™ headers (2x6)
- 8 x Slide switches and 7 x Push button switches
- Stereo line in/out, Headphone and Microphone input
- Xilinx XADC header, supports 4 analog inputs
- On-board USB JTAG programming port and ARM Debug Access Port (DAP)
- size: 160mm x 134mm

2.4.2 Adapteva: Parallella

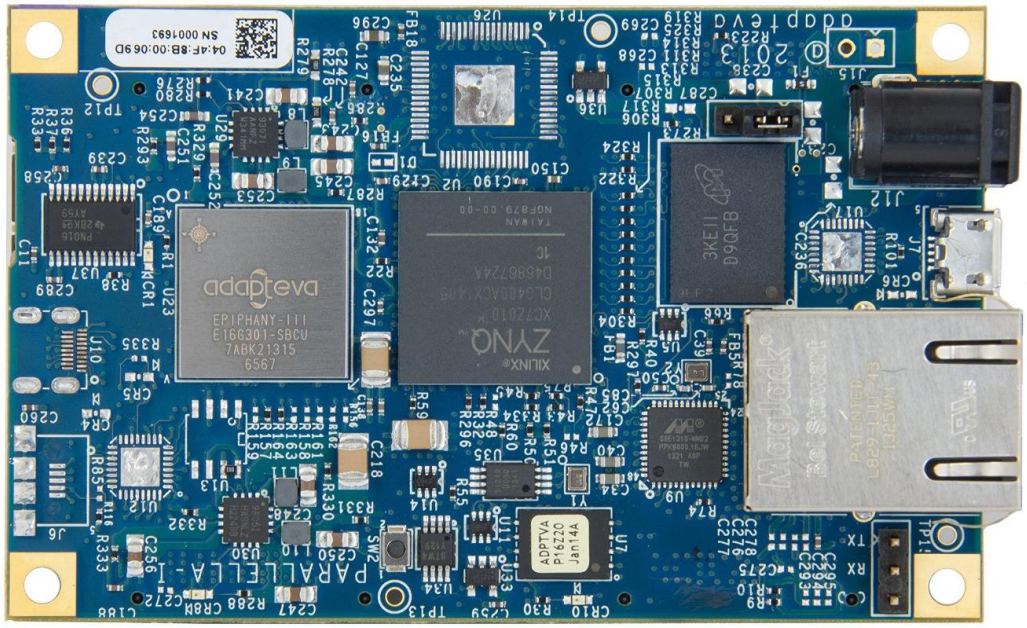

Figure 2.6: Parallella, from [4]

The used Parallella Board does not have the GPIO (Samtec Connectors) populated, as it is a pre-order version. It has the following specifications, complete description in [17]:

- Zynq-Z7010 with Dual-core ARM A9 CPU
- 16-core Epiphany Co-processor
- 1GB DDR3 RAM
- 128Mb QSPI Flash
- Micro-Secure Digital (SD) Card
- Gigabit Ethernet
- micro HDMI
- 2 x Universal Serial Bus (USB) 2.0
- Up to 48 General Purpose Input/Outputs (GPIO) signal
- Linux Operating System
- Size: 55mm x 90mm x 18mm
- Weight: 38 grams

2.5 ER-4

As to enhance, an experimental application of such FPGA SoCs and also to introduce the Infrastructure that originated the goals of this thesis, the ER-4 will be presented. It's composing hardware is Fabricated from a Xilinx FGPA chips competitor, namely Altera. Although, the specifications of the FPGA chip are similar to the ones already introduced from Xilinx.

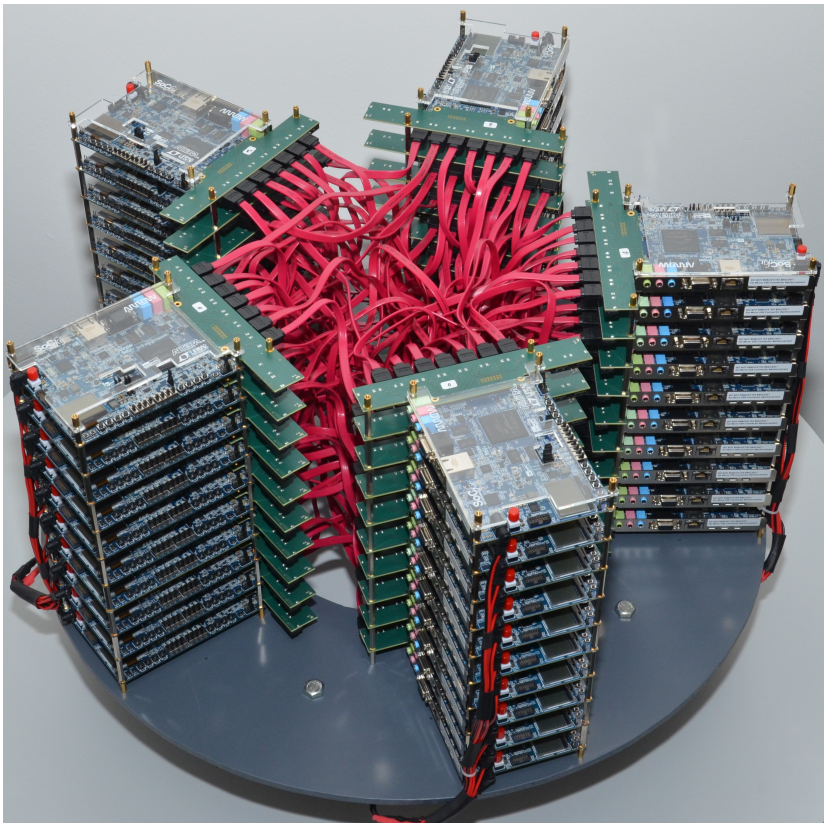

Figure 2.7: ER-4 Set up, from [5]

The ER-4 is an experimental parallel computer set-up, consisting of a network of FPGA chips, disposed in a pentagon, see Figure 2.7.

The ER-4 integrates 50, Terasic fabricated, SoC boards, based on the SoC chip 5CSXFC6D6F31C8N from Altera.

Each FPGA chip, that is based on the Cyclone-V, integrates two ARM cores, which have access to two separate 1GB RAMs and various other interfaces, as well as 9 high-speed serial interfaces. Also other peripherals for the ARM subsystem, like Ethernet interface, USB master and slave interfaces, SD card, Video Graphics Array (VGA) output, LEDs, switches, a small LCD display and audio connector for input and output.

The serial interfaces interconnect the network of the FPGA boards through 8 interfaces per node. They are connected between the other FPGAs with Serial AT Attachment (SATA) cables, in a non-standard network topology.

The structure of the network that connects the 50 FPGA nodes uses the Hoffman-Singleton graph with a diameter of 2. More reference to the Network topology and other information can be viewed in [5].

2.5.1 Configuration of the ER-4

To configure the ER-4 as a functional parallel computer, some developments have to be taken care.

Starting with an infrastructure that gives room to the execution of applications. One would like to fill the FPGA parts in each node, with as many soft processors as possible, depending on the application. The FPGA also handles the communications between the soft processors, the ARM and the other nodes.

Supposing that different FPGA configurations could be used by different applications, one could store this configuration files on the SD card of the SoC.

To control the parallel computer, an external operating system could command the operations that are performed by using the Ethernet interface.

As so, the ARM on the FPGAs could be equipped with a simple runtime system to support the Ethernet interface, program download, SD card access and FPGA configuration.

The FPGA part must be suited with an infrastructure to handle the external interfaces, common to all nodes, and that is constructed based on the needs of the application.

An host port in the FPGA part would be controlled by the Ethernet, with a communication interface through the ARM. Also the video and audio ports could be accessed through the host port.

The soft processors configured in the FPGA also need an interface to access the chip external memories.

The implementation of this architecture is suggested by[5].

2.5.2 Pi-Nets Applications

The type of the applications that were designed to run on the experimental computer, was given the name of Pi-Nets. The Pi-Nets application is executed on each node, controlling the execution of all individual processing units. It will be executing code on the ARM cores and some logic on the configured FPGA.

2.6 PiNets Runtime System

To assist the PiNets Applications that are to be executed on the FPGA and ARM systems, the standalone PiNets runtime system was created.

With that in mind the objectives of the runtime system are the following:

- Providing a file system and the corresponding access functions for an SD card
- Providing routines for FIFO-based Ethernet communication
- Treatment of the Ethernet packets addressed to the runtime system

  This includes:

  - Reply to version number requests
  - Initial configuration and launch of a PiNets application
  - Sending a download request of the application, if necessary, and save the downloaded data in the SD card
  - Programming the FPGA
  - Writing configuration and status data in to the RAM memory
- Specifying a memory allocation for the system

This Runtime System is used as base to some topics of this Thesis. This Software was initially created by an other student, he developed the topics that will be explained in this chapter as documented on [18]. This system was designed for the ZedBoard using the Xilinx Vivado tools.

The focus of this thesis is to proceed with his work and add the necessary functionality to it. By using this Runtime system as a software base of the Zynq no other operating system is necessary, as it provides the necessary functions. It runs standalone, in a Bare metal configuration, and provides a good base to design the remaining components of the infrastructure.

The key features of this runtime system will now be presented, they will support some components of the infrastructure.

2.6.1 PiNets Application Support

The main usage of such a runtime system is to assist the PiNets applications to be executed in a complex net of FPGA and ARM systems, the ER4. The SoC's are mapped with its own unique hardware connections, but to communicate with the Host Desktop the Ethernet connections is used. Such that the launch and management of the PiNets applications that can be executed by the parallel computer can be handled.

There are three files that constituted the PiNets application who is executed in the SoC, they are composed by the FPGA configuration file, the ARM0 instruction code and the ARM1 instruction code.

Both ARM core units will have its own code to execute, that means that both ARM cores are processing code separately, in bare metal configuration. The system was designed to operate this way. Special care has to be given to the caches for coherency in the data.

PiNets Communication Protocol

The PiNets system defines a communication protocol for the Ethernet frames, defining different types of frames. The used transport layer is RAW Ethernet. To the raw Ethernet header a PiNets header was added, this header has the structure defined in Table2.1, where the PiNets key is a random number defined as 0x12B9B0A1.

	Raw Ethernet Header			
	Destination MAC	Source MAC	Ether Type (0x6712)	...

	PiNets Header			PiNets Data
...	Zynq MAC	PiNets Key	Message Type	...

Table 2.1: PiNets Ethernet frame received by the runtime system

As for the defined Message types that can be used and their structure, are represented in Table2.2.

PC request, via broadcast:

0	PC version	0	Source Computer name	0

Replay to Message 0 from PC:

1	Software version	0	Zynq name	0

Download PiNets Application:

2	0	0	App Code	Sequence	Data

Write Status Data to RAM:

3	N	0	App Code	Address	N Words

Start a PiNets Application:

4	App Version	0	App Code	0	Source Number	...
...	App Name	0	List of Ethernet Addresses			

Request to Download Application, replay to Message 4:

5	App Version	0	App Code	0	Source Number	App Name	0

Signal readiness at start up:

6	App Version	0	App Code

Traffic with FPGA's Host Port:

7	data

Table 2.2: PiNets Message types structure

2.6.2 SD card access

The runtime system provides access to the SD card, by initialising a file system in the SD card. It also provides functions to handle the files contained in the SD card.

The PiNets applications are stored in the SD card prior to execution. They can be saved to the SD card with a specific Ethernet message or simply by copying the files to the SD card with a Desktop computer.

The application files that are stored in the SD card follow a special naming convention that must be used such that the files can be read. The naming convention is used as follows:

- FPGA configuration file: app[AppCode]_v[Version]_[Name].bin
- PiNets Instruction Code ARMx: app[AppCode]_v[Version]_[Name].ARMx.elf

Where x can be 0 or 1, corresponding to core0 and core1, respectively. Both versions must be present on SD card, to run a PiNets Application.

The SD card must be formatted with FAT32, such that the implemented file system can manage and access the SD card. The file system is based on the FatFs, a generic FAT file system module that can be incorporated into microcontrollers. This implementation delivers all necessary functions, except to the hardware access, but this is handled by the Xilinx Board Support Package, with the library "xilffs". The most usual user functions are available for SD card access, they are f_open, f_close, f_read and f_write.

2.6.3 Ethernet Interface

The Ethernet hardware is initialised by the runtime system, with a data rate of 1Gigabit/s. In order to process this data rate without overflow, the DMA burst size is set to 16 bytes.

The same padding is also provided in the frame buffers.

The MAC address is set by the array defined as *zynqMAC1*.

The Ethernet hardware of the Zynq makes it possible to define a type ID to compare the incoming Ethernet frames and setting a flag accordingly. This function is used to filter the incoming PiNets frames, without having to load the actual frame from the RAM.

Incoming packets are processed by an interrupt service routine. Quick answerable inquiries are handled directly by the interrupt, other complexer requests are forwarded to the FIFOs of the runtime system, to not block the program flow in the interrupt routine.

Incoming and outgoing frames are stored in a FIFO, to buffer the communications. A FIFO is defined as a structure. An Ethernet frame structure contains the data and length of a frame and an instance to the interrupt controller. When an interrupt occurs the respective FIFO will be filled. To ensure mutual exclusion in critical sections, the corresponding interrupt is disabled.

An Ethernet frame structure can not be created by the user, to administrate such structures only pointers are to be used. The user can borrow such structure from a FIFO in order to manipulate it. Then the structure is returned back to the runtime system.

The user can interact with this Ethernet frame structures by using the following functions:

- u32 ethernet_getReceivedFrameCount(u32 fifoNumber): Gives back the number of elements, which are in the User FIFO fifoNumber.
- ethernetFrame* ethernet_borrowReceivedFrame(u32 fifoNumber): If the user FIFO fifoNumber is not empty, an item is taken from it, if its empty a null pointer is returned.
- u32 ethernet_getAvailableEmptyFrameCount(void): Returns the number of available FIFO elements.
- ethernetFrame* ethernet_borrowEmptyFrame(void): Takes an item from the available FIFO, when not empty. If empty, a null pointer is returned.
- int ethernet_returnFrame(Ethernet frame * frame): Returns the borrowed structure to the FIFO entries. Returns 0 if everything went well, otherwise a 1 will indicate a bug.
- int ethernet_sendFrame(Ethernet frame * frame): Sends the created structure to the transmit FIFO, to send through Ethernet. Returns 0 if everything went well, otherwise a 1 will indicate an error.

The incoming Ethernet packets that are addressed to the runtime system are processed by analysing the PiNets Ethernet Header type, such that the corresponding function that will handle the interrupt can be called. These functions can be called to answer to a version number request, to write configuration or status data in to the RAM and to start or download PiNets applications.

2.6.4 Multiprocessor system interactions

The runtime system starts both the ARM core0 and the core1, although the main system runs on the ARM core0.

It is also from the ARM core0 that the SD card is accessed and the Ethernet communication is handled.

The available on-chip memory (OCM) in the range of 0x00000000 - 0x0002FFFF is marked as uncached, and is preferably used for communication between the two ARM cores. To avoid

problems, the ARM core1 uses at start up of the program only the L1 cache. The L2 cache is allocated only to the ARM core0. In particular, it must be ensured that the ARM core1 does not flush or invalidate the L2 cache without performing a synchronisation with the L1 cache.

2.6.5 DDR Memory allocation

The memory allocation for the various functions and instruction code can be viewed in Table 2.3.

Entries	DDR address	Size
_standardFunctionPointers_ARM1	0x10C80000	0x080000
_standardFunctionPointers_ARM0	0x10C00000	0x080000
_piNetsProgramCode_ARM1	0x10700000	0x500000
_piNetsProgramCode_ARM0	0x10200000	0x500000
_FPGAConfigFileTempStorage	0x0FA00000	0x500000
_EthernetFramesStorage	0x0F018000	0x9E8000
_txBDList	0x0F00C000	0xC000
_rxBDList	0x0F000000	0xC000
_ARM1_standardStack		0x100000
_ARM1_standardHeap		0x100000
_ARM1_ProgrammCode	0x02000000	
_ARM0_standardStack		0xE00000
_ARM0_standardHeap		0x100000
_ARM0_ProgrammCode	0x00100000	

Table 2.3: DDR Memory allocation, in read uncached address space

The entries are transferred directly to the linker script, such that the generated .elf file contains the pointers to address the entries.

Some remarks to the presented entries are given below, where in ARMx the x can be 0 or 1 corresponding to the desired ARM core:

- _standardFunctionPointers_ARMx: Where the addresses of the functions are stored, which are made available for the PiNets program by the runtime system environment.
- _piNetsProgramCode_ARMx: The executable PiNets program code.
- FPGAConfigFileTempStorage: Where the FPGA configuration file is temporarily stored, for FPGA configuration.
- _EthernetFramesStorage: The Ethernet frame structures that were provided for communication.
- _rxBDList: List with the received buffer descriptors is stored here.
- _txBDList: List of the transmit buffer descriptors is stored here.
- _ARMx_standardStack: Stack of the runtime environment. Can be used by the PiNets Program.

- _ARMx_standardHeap: Heap of the runtime environment.
- _ARMx_ProgrammCode: The runtime system's executable code.

2.6.6 Boot and program handoff

As the Runtime system was designed using the ZedBoard as development hardware one can boot it with different methods.

To allow such boot options, a FSBL must first be created. This code is generated with the Xilinx Software tools, specific to the designed and configured hardware. Also using the Xilinx tools the FSBL can be attached to the runtime system and so create a binary file that is capable of booting the Zynq. The boot source, in the case of the ZedBoard, can be configured with jumpers, and it's possible to use SD card, QSPI and JTAG to boot it.

As the Zynq is booted, and the handoff to the Runtime system is concluded, a flag in the OCM is set to the ARM core1. By setting this flag the ARM core1 will wake up and answer back. Some initial configuration is performed, regarding the caches, in which the core1 uses only the L1 cache.

The next step is to initialise the SD card, in this same step the file system of the SD card is created, such that it is ready to use. The pointer of the file system is passed in the flow of the program to the other functions.

The configuration of the Ethernet hardware will follow, configuring the PHY and the frame filter. To properly configure the PHY a 4 seconds delay is given, where the processor will be sleeping. In this step also the interrupt controller is configured and the FIFOs are prepared for readiness.

As this initial configurations are concluded, the program enters in a state where it waits for a Ethernet message of type 2, to start a PiNets application. If other PiNets messages are received they are handled by the interrupt controller directly. If a message 2 is received the program will analyse if the required Application is stored on the SD card, if the application is not present, a message 5 is constructed to request the application files, and the program will wait for its arrival. If in the other case, were the application was already present on the SD card, the runtime system will start its configuration immediately.

To start a PiNets application firstly the runtime system will configure the FPGA, by loading the configuration file stored on the SD card to the RAM memory and executing a function that configures the FPGA from the PS. After the configuration was executed, the program code for the ARM core0 is loaded in to the pointed RAM address, the same is done to the program code of the ARM core1.

At last, when the FPGA is configured and the PiNets Application code is loaded in to RAM, the runtime system signalises the ARM core1 to handoff the execution to the PiNets Application. And as for the ARM core0 the runtime system transfers its execution to the piNets Application, by pointing to its core0 Application code start address.

Chapter 3

Design-Flow and Support-tools

In this chapter, the followed design flow will be presented, as to design a FPGA project one has to take care of synthesising and implementing the design. Also the necessary tools for the implementation of this thesis will be presented, including all Hardware support tools and other development Software tools.

3.1 Design Flow

When a digital system is to be designed for a FPGA device, the Designer has to take care of four major design steps, the HDL design, synthesise the design, implement the synthesised project for the specific hardware board and, at last, programming the device and testing its functionality.

Each of these steps have its own verification scheme, by simulation, testing or other analysis. The system can be simulated at a behavioural, functional and temporal level, enabling the designer to discover errors at a early stage. If improper results were verified the designer can return to each of the steps to correct the failure and advance when the proper functionality has been achieved.

Adding a PS to the FPGA , involves some extra steps, that are to be taken care in order to make the PS available. The PS can be regarded as a complex module inside the FPGA, with lots of configuration possibilities. To use the PS, the configurations of its hardware must be defined. When this additional step is achieved, the normal FPGA design flow is applied, as the configurations of the PS does not concern the normal verification methods and are so bypassed from it. After a valid design has been implemented, a hardware platform can be created. This hardware platform will serve as base for developing the Software that runs on the PS.

In Figure 3.1 are the steps of the design flow represented. It can be verified how the Steps are linked to each other, concerning the design and its verification, sub divided in a Hardware and Software part.

Each of these processes will be described in more detail in the following subsections.

3.1.1 HDLs - Hardware Description Languages

A HDL is a computer language which is used to describe digital logic circuits. Using a HDL one can describe the structural and behavioural description of a circuit.

The HDLs emerged as an abstraction layer for the creation and modification of logical circuits. The designs performed with a HDL can also be simulated, and afterwards with

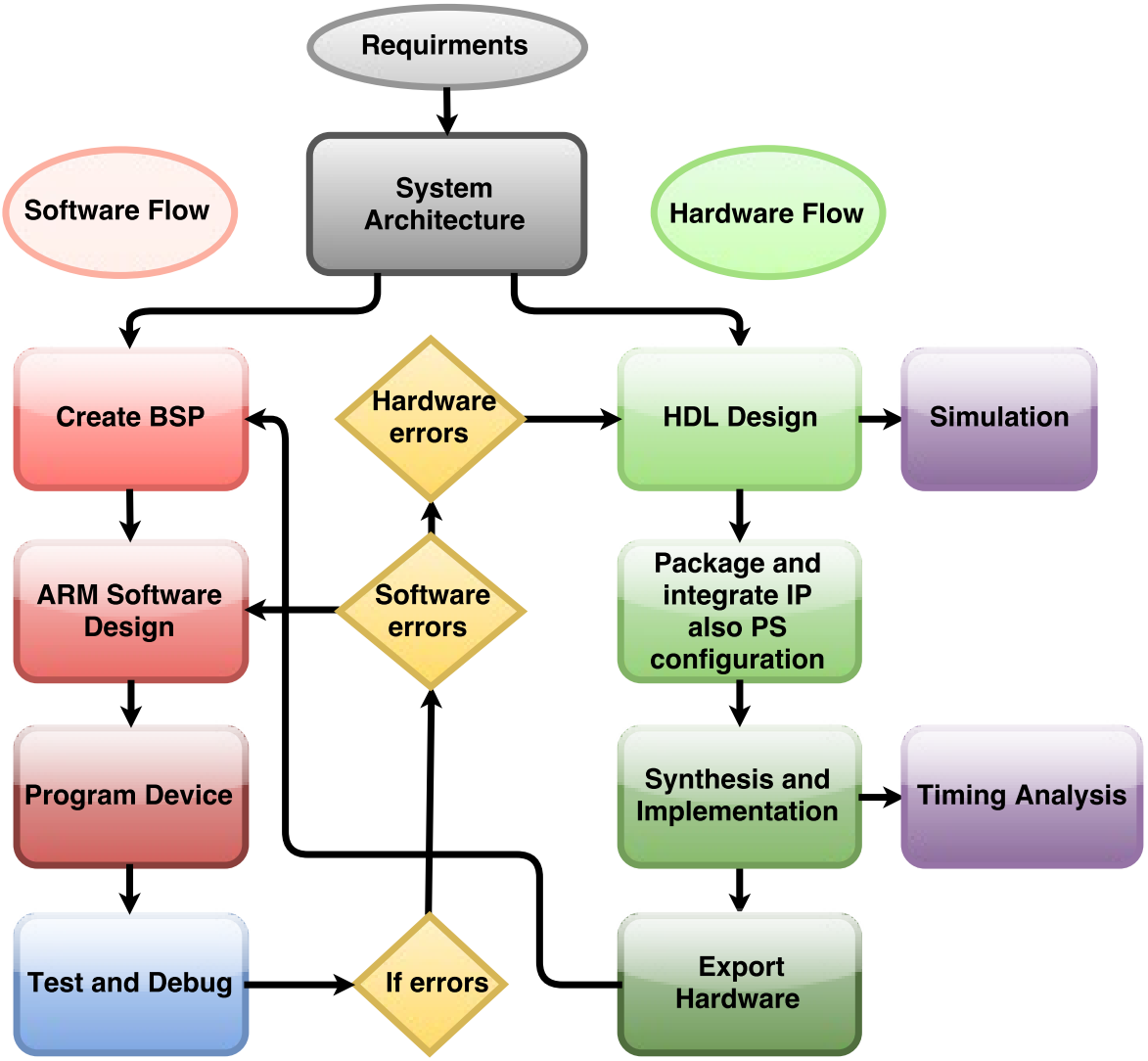

Figure 3.1: Design flow

the help of some Software-tools be synthesised and implemented. The HDL design is for the designer the fundamental tool to describe the logic circuits in the FPGA. Within the available HDLs, the two most popular languages are Verilog and VHDL, but other languages are also used. In this Thesis, the designs will be carried out using VHDL. A brief introduction to this language will be given in the next section.

3.1.2 Synthesising the Design

After the creation of a HDL that is free of syntax errors, and that all the necessary simulations have been performed, the synthesis can be accomplished. The synthesis is the software process were the code in the HDL file is analysed and a translation in to logical components is performed.

The synthesis generates a netlist of the synthesised project, this netlist contains a description of all the needed hardware components, the interconnects and the names of the I/Os that are to be implemented.

The netlist can be generated by multiple software tools, and multiple optimisation targets

can be selected for translation. When performing a synthesis, the software will carry out an optimisation of the HDL, such that the optimisation strategy is respected, using the smallest possible number of hardware resources.

It is possible to write a HDL code that is correct in terms of syntax, but that is not possible to synthesise. When such a code is designed, it's only possible to use it for a behavioural simulation. The designer must take care, when designing with HDL, that the description is physically possible to implement.

3.1.3 Implementation of the Design

As soon as the synthesis is accomplished and a valid netlist file is generated, the project is ready to be implemented. The implementation of the project is divided in to three sub processes, the translation, the mapping and the place & route.

Where the translation is the process in which all the information regarding the synthesised netlist and device specific libraries are imported and prepared for layout. In this same process an optimisation is accomplished, based on the user specified optimisation rules. The design is also compared with the target hardware, for compatibility matching and to check device associated rules.

In the next step, the logical components specified in the netlist are mapped to the available components in the hardware device, by mapping also the IOs to the physical hardware ports. If the device does not contain sufficient resources to implement the project, the process will terminate abruptly with an error.

As last step of the implementation, the mapped netlist is outlined into the FPGA primitives, by placing the modules or logical components where the design gates will reside. This process is complemented by the route, where the physical routing of the various logical components is fulfilled, by interconnecting them according to the mapped netlist.

At the end of the implementation, some reports are generated, regarding the timing analyses and also the consumption of resources. An important highlight is the timing analyses, in which the maximum delay spread between components is obtained, this delay can lead to failures or malfunctioning in the configured logic. Using tools that are not available for this project its possible to determine the maximum allowed clock frequency for the implementation.

3.1.4 Programming the FPGA

Once the design is implemented, the FPGA configuration file can be generated. The implementation creates all necessary files for the generation of a device programming file. This file that is in a binary format contains a bit stream ready to be loaded to the FPGA and to configure all the logic blocks and its interconnections.

There are multiple ways to program the FPGA, one of them is by plugging in a JTAG cable. In this configuration the device's software tools assist the configuration. With this method one has the biggest control of the device, as we can analyse the logic directly in the internal chip, but, when powering off the FPGA, the configuration will be lost and one has to reprogram the FPGA.

Another possibility would be to load the configuration file to a SRAM, and configure a boot source to configure the FPGA, in a way that it would automatically start up when a power source is attached. With this method the FPGA also loses its configuration at power off, but it will automatically configure it self at power on.

3.1.5 Hardware Platform for the PS

With the implemented design its possible to generate a hardware platform for the PS. This statement is just valid if the design covers the essential PS configurations.

Depending on the software tools, the hardware platform may be generated and linked to the Software development kit. An important step that the hardware platform enables, is, by using the SDK, generating a Board Support Package. This BSP contains important functions to handle some of the configured hardware, and it also contains functions to boot the Zynq. This platform can be used to develop software for the PS.

3.2 VHDL

As referenced, the HDL will be designed using VHDL. Using this language one can describe logical digital circuits. The VHDL is a very popular language, widely used and one of the most used languages to describe digital logic circuits.

The VHDL can be used to describe the structure and the behaviour of a digital circuit. The structural description allows to divide the digital circuit in multiple modules, that compose a bigger module, and the bigger module can be integrated in a even bigger module. The only limitation of the dimension of the circuit is the capacity of the Device, but considering the VHDL language, one can produce circuits without dimension limits. The behavioural description defines how the circuit behaves concerning the input signals and the produced output signals.

To simulate a designed HDL, one can write a testbench, using, as well, the VHDL language. This testbench combines a structural and behavioural description attached to the top of the HDL that is to be simulated. Verifying all the possible input possibilities and observing the resulted output.

An entity is, in VHDL, a module that represents a logical component or a description of the module's interconnections. In Listing 3.1, a generic VHDL entity is given, as well as its architecture.

On the first part, where the entity is defined, the input and output signals are declared, and also constants can be stated. On the second part, the architecture of the defined entity can be described.

The information provided in this section is an introduction to the VHDL, for a more complete understanding consult [19] and for the language's syntax refer to [20].

3.2.1 Structural description

A digital system can be constructed as a block design, where multiple modules are interconnected. Each module consists of some logic, and defines input and output ports. These ports are the interface to the other modules or to chip external peripherals. By describing the structural description, one can link the modules together as in a block design, but following the writing syntax defined by VHDL.

3.2.2 Behavioural description

At the behavioural description, the logical behaviour of the module is defined. The VHDL defines some procedures that can be declared, such as processes and functions, to define the logic of a module. A process can be used when a sequential analysis of the logic is to be performed. In the context of a simulation, a process is analysed when the signal that was

```
entity <entity_name> is
generic( -- generic assignments
 <generic_name> : type [:= <initial_value>]
);
port( -- port interface list
  <signal_name>: in|out|inout|buffer type
);  end [entity | <entity_name>];

architecure <architecture_name> of <entity_name> is
[
--signal declarations
signal <signal_name> : type;
--constant declarations
constant <constant-name> : type := <initial value>;
...
--combinatorial statements
--sequential statements
...
] end architecture;
```

Listing 3.1: VHDL entity and architecture syntax

declared at the sensitivity list, has changed. It's usual that this signal is the clock source, such that the response to the inputs are analysed at each clock cycle.

An example of a process routine is represented in Listing 3.2.

3.3 Software Tools

As this thesis is a CAD based project, a considerable amount of tools are needed to develop such a project. By developing the project with a Xilinx based FPGA, the tools that can be chosen to deal with this device are considerably lower. Another aspect is the availability of such tools, that are mostly not for free. The manufacturer Xilinx, offers a free WebPack of both its most popular tool kits, Vivado and ISE. The complete version of both these tools is available at the institute.

Besides the FPGA tools, some tools are also used to control some aspects of the medium. Starting with the picked operating system and going through all the tools that are needed to communicate through some interfaces.

3.3.1 Operating System

The FPGA tools from Xilinx are compatible with multiple Operating systems, and they can be installed on multiple versions of Windows and Linux. The Windows is a very stable OS, providing the user with a very friendly and uncomplicated user interface. But, for this project, a Linux distribution was chosen. It offers a more versatile Software, where the user

```
process_name: process(clock_signal)
constant constant_name : std_logic;
variable variable_name : std_logic;
begin
   if rising_edge(clock_signal) then
      if(sensitive_signal = '1') then
         output_vector <= constant_name;
      else
         output_vector <= variable_name;
      end if;
   end if;
end process;
```

Listing 3.2: VHDL process example

can control every aspect of the OS. In addition, the development and execution of applications on Linux is less complicated than on windows.

The project was developed under the operating system Ubuntu 14.04. This particular distribution is a long term support and, from daily use perspective, one can say that it is very stable.

3.3.2 Xilinx Vivado Design Suite

To be able to design a project on an FPGA, tools to help performing the design flow are necessary.

Between the two referred Xilinx tools that could be used to develop this project, Vivado was the chosen one. This decision was made because of the following reasons:

- The Vivado Design Suit is the most recent Xilinx tool to design FPGAs, and the only one further supported.
- Supports the 7-series FPGA, in which the Zynq makes part of this group.
- Handy user interface, designed to enhance the development time.
- Bigger and enhanced IP repositories.
- Some projects that where designed for the Zynq are in the Vivado format.

The Vivado tools are very complete in its capabilities, and can fulfil all basic needs of a FPGA designer. The designing of the VHDL code may not be done by the text editor offered by this tool, as other text editors offer a better assistance to facilitate the designer. As for the simulation, this can be done well enough with the Vivado tools. When it comes to synthesise and implement the project, this is the right tool to use, it was build for that purpose. Also for the configuration of the FPGA afterwards it is a good choice.

The Vivado tools are based on a block design perspective. Where all modules present in the project are linked in the block design entry. The tool's biggest advantage is the way it configures the PS in an intuitive graphical interface. If a PS is to be used, this module is to

be added in to the block design, and configured respectively. The project is build with the designed modules in turn of the PS. Each module, that can be a user created HDL or some IP core logic, has to be packaged to be included in the block design scheme.

With the Vivado tools one can control all the aspects of the development flow, offering a intuitive graphical interface, to access all the existing features. The default layout of a Vivado project is represented in Figure 3.2. The default desktop is divided into four major windows. The window on the left is divided in to seven sub menus, they control the actual development phase, that the developer is working on. This window is the only one that does not change its content, all the others, depending on the selected section, can change their content. The opened section is identified on the top of the right bigger window. Inside this bigger window there are three other smaller windows, in which the one on the left shows all the files and sources associated with the project, in a hierarchically manner, the window on the bottom is used to insert Tcl commands, view Logs, error messages and others, and, the final window on the right, is very dependent on the opened section, it will change its content depending which stage the developer is working.

All these windows can be resized and moved around as the user's will.

To the sections will be given some description as they will become of major importance in the development phase.

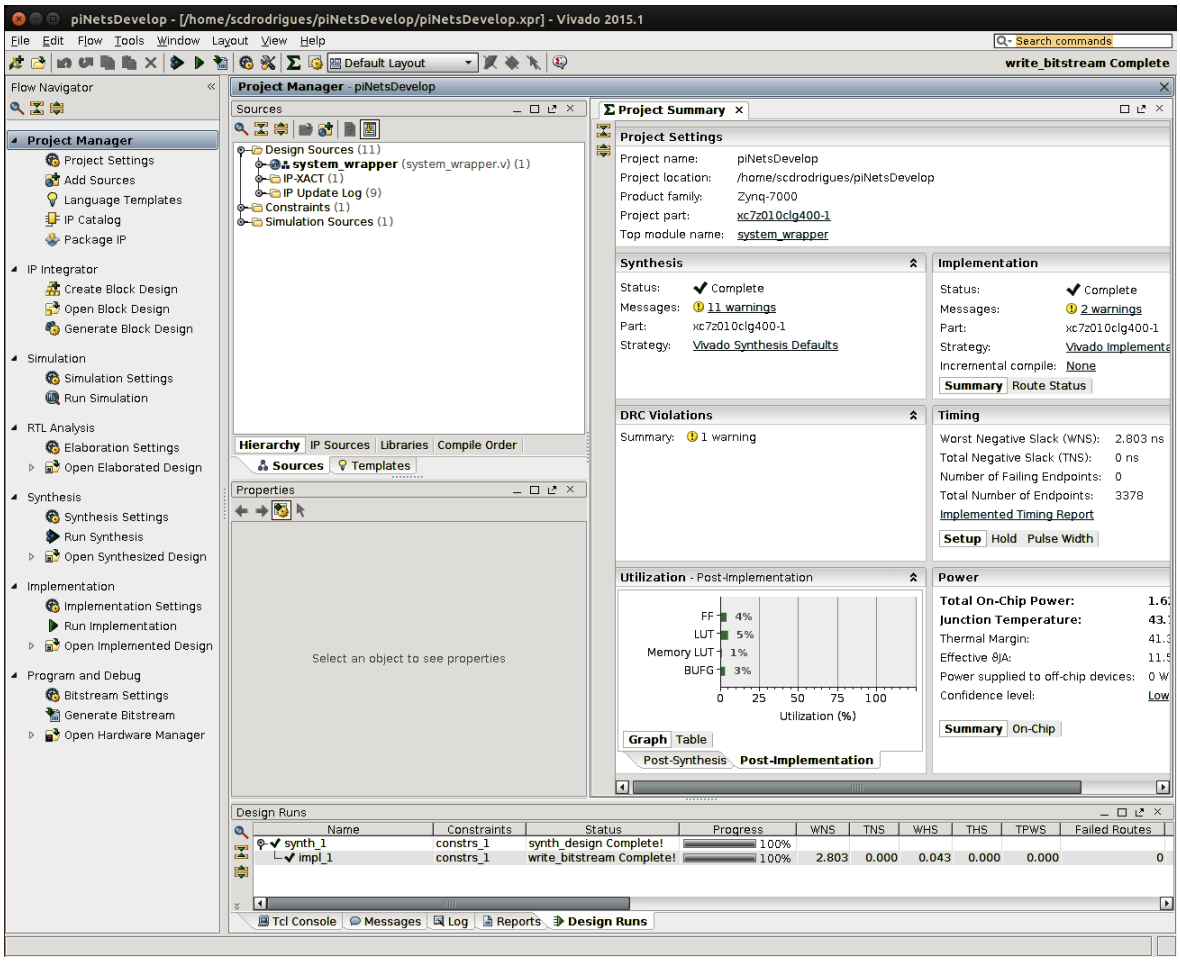

Figure 3.2: Default desktop of Xilinx Vivado

- Project Manager - When opening Vivado this section is displayed. This window enables a general view of the state of the project. In this section one can configure general settings of the project, like the Project Device or the target hardware language. It also displays reports of the project when tasks were performed. Through these Report it is possible to identify design problems and its origin. Its also possible to verify the consumed FPGA resources.

- IP Integrator - This is the main developing window, where the block design is presented. Logic blocks can be added from the IP repository. The IP repository contains all IP core sources from Xilinx and the modules that have been packaged by the user. The PS can be added in this block design window as a IP core and also be configured. When the block design is fully designed, it can be validated, followed by regenerating the sources, such that the integrated IP cores are loaded to the project sources.

- Simulation - This section enables the developer to initiate a simulation of the selected Top project.

- RTL Analysis - With this option its possible to design a floor-plan, elaborating how the components are interconnected and the IOs placed.

- Synthesis - When the project is fully designed this option can be used to synthesise the project. The optimisation settings can also be properly configured. After the synthesis is complete, the synthesised project can be opened and its reports can be analysed.

- Implementation - After a valid synthesis has been generated, the project can be implemented by using this option. Once again the optimisation settings can be adapted to the required needs. If the constraints were not added to the project sooner, they can be constructed at this point. All the external pins of the device are presented in this section, and it's possible to link to the project's IOs. The implemented project can be reviewed by analysing the generated reports.

- Program and Debug - The last section is used to generate a FPGA configuration file and to program the device. If a logic analyser block was added to the project, signal debugging can be performed.

On the window's top menu are numerous options left, were the user can open some Xilinx FPGA tools. One of the most important ones, is to create and package custom IPs. It is also possible to export the designed Hardware Platform to the SDK platform.

The used version of the Xilinx Vivado tools was the 2015.1. For more information of the Design Suite please refer to [21].

Xilinx IP Catalogue

The Xilinx IP Catalogue, standing for intellectual property, is a repository of logic blocks designed to be configured by the user as its needs. The catalogue has a vast variety of entries for all possible design purposes. They are designed to fit the needs of the designer by enabling a high configuration possibility. Using some IP cores in the design, enables a faster development of the project, and more time to concentrate on the key modules.

Simulation

The simulation process is of most importance to design a digital system. As the implementation of the project is a very time consuming process, many steps have to be accomplished to get a testable file and to program the FPGA. If simulation is performed to correct errors at the earlier steps, one can save much development time. It's very simple to make mistakes when designing with HDLs, and many errors can be avoided with a simple simulation.

The Vivado Design kit contains a HDL simulator, which provides a satisfactory simulation environment, where one can verify the behaviour of the written HDL, through a graphical timing chart of the IOs signals. Analysing the signals at any point in time, allows to determine if the intended behaviour is being applied, facilitating, this way, troubleshooting.

3.3.3 Xilinx SDK

By introducing in to a FPGA building block a PS, one has the need for a specific development program for the PS. In this case, the ARM needs to by fully supported on the software side.

The Xilinx Vivado tools incorporate the Xilinx SDK, that is a software development platform based on the eclipse platform, but worked around to support the ARM on the Zynq.

To start a design on SDK, the PS needs to be configured with the Vivado main platform, and the project needs to be implemented. With the implemented project, its possible to export the created hardware platform to SDK. Once the hardware platform is present, the SDK can be launched, and the respective board support package can be generated. The creation of a totally new project, that can be executed by the ARM, can at this point be initiated. The SDK platform offers some example functions that are ready to use, and are capable of demonstrating some uses of the Zynq. They also point to some important function contained in the BSP. Every new created project is linked to the sources provided from the selected BSP.

The Xilinx SDK also incorporates a tool to debug the PS. It is possible to configure a live session, using a JTAG connection, and verify the instructions that are being processed. The register's values and some memory content can also be viewed. Making it a very important tool to help debugging.

In Figure3.3, an example of a bare metal project, created in SDK, can be seen. Hear, one can see that a hardware platform is included, containing a important function to boot the Zynq, as the *ps7_init.c*. On top of the hardware platform were created the BSP. As we are dealing with a bare metal project, two BSPs had to be created, as both ARM cores are treated separately.

For more information about the Xilinx SDK please refer to [22].

3.3.4 Eclipse

Similar to the SDK work environment, but for the development of other platform software, a program delivering a development work space is necessary for this project. As the SDK has compilation libraries for the ARM processor, one can not use it conveniently to develop software for other architectures.

The Eclipse Platform was found to be a good option. One can integrate multiple software development tools supporting many platforms. Also, this software has a open source licence. Its development environment equips the user with a well integrated and a very nice user interface. It's relative easy to use and organises the project in a good manner.

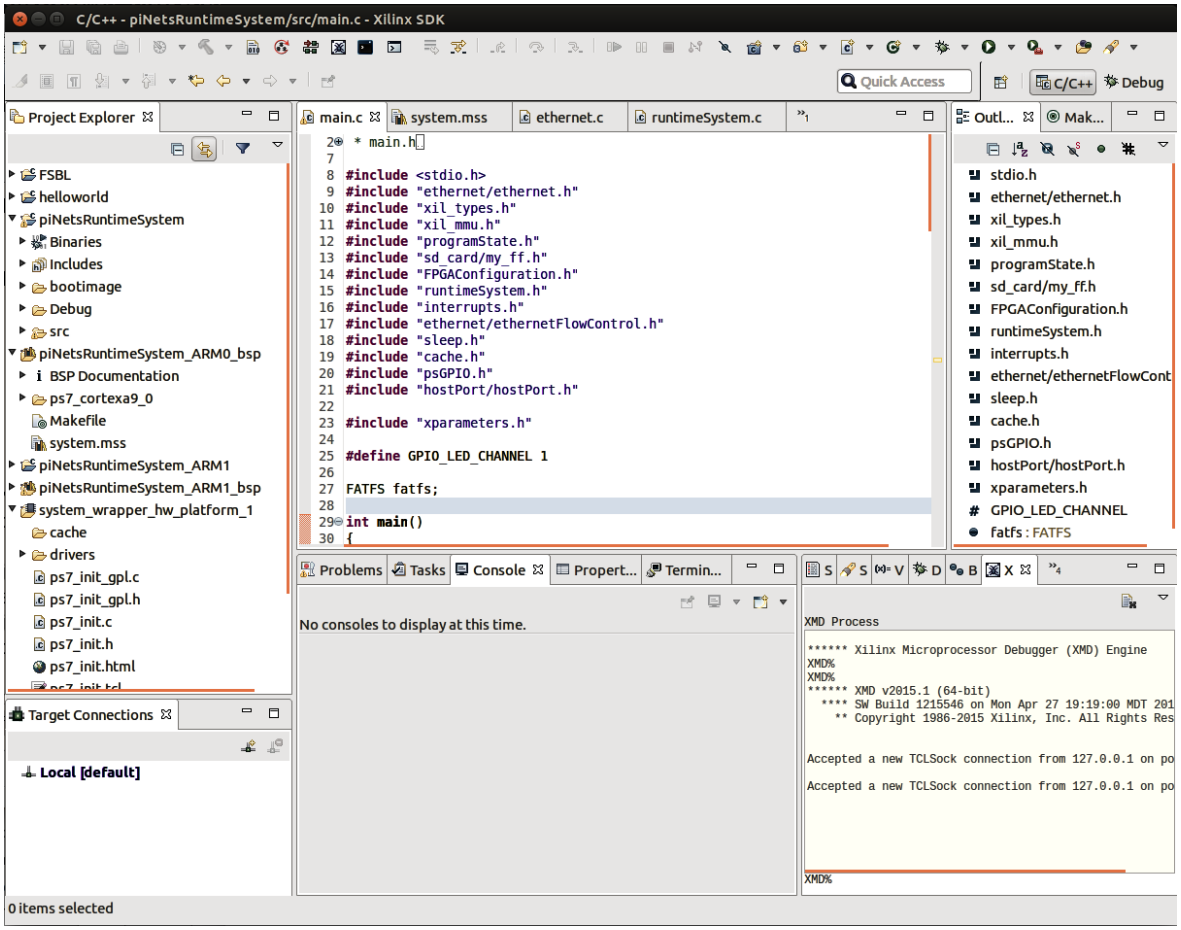

Figure 3.3: SDK default desktop

Multiple development tool-kits can be installed to support a vast variety of programming languages and also some hardware tools. One of these tool-kits was necessary to be installed as it supports the compilation of the Epiphany executable, beyond other chip support.

The Epiphany SDK's, is based on Eclipse, but it contains other resources, like a local debugger platform and a Epiphany core simulator among others. All its features and general information can be found on [23].

3.3.5 CuteCom

To communicate with the hardware device a UART turns to be essential, at least at debug stage. A program that can send and receive data from a UART interface and has a open source license is the CuteCom, from [24]. With this program one can receive data both in ASCII or HEX formats, and also send in both formats data to the UART interface. It can also be configured according to the UART specification, with different numbers of data and stop bits.

The UART is linked through a USB cable.

Chapter 4

Architecture

In this chapter the features that will be added to the runtime system will be presented and the architecture of the Operating Infrastructure will be further analysed.

4.1 Operating Infrastructure

Part of the operating infrastructure's purpose is to provide the necessary hardware environment in a way that the components in the PL can execute applications. This hardware environment are digital circuits, programmed in the FPGA, that grant access to external hardware components, such as memory, video output and also other complexer components like a co-processor.

Because we are dealing with a system, which contains both PL and PS, also the software environment has to be considered. It must be properly configured to run applications.

The main objective is to run Applications on the PL and on the PS. In this way, the infrastructure is not entirely implemented on the PL, it also involves some ARM software on the PS and also a server on a PC to input and output test data.

The infrastructure's components are represented in Figure 4.1, the services that it covers are as follows:

- A Host Port interface with a data port from the host PC to the components on the PL.
- A video interface to output text on an external display.
- Interface to the host PC supporting IO for test data, FPGA configuration and application download and start.
- An interface to the memory to be accessed by the hardware components on the PL.
- Interface and start of the Epiphany co-processor chip.

Combining the services together generates the operating infrastructure, containing the basic modules that are needed by almost every application that is to be executed on the Zynq.

To understand how the infrastructure is constituted, how it is attached to the Zynq and to explain the features that will be integrated in to the Runtime system, please refer to the next sections. The architecture of each of the referenced services is going to be analysed further.

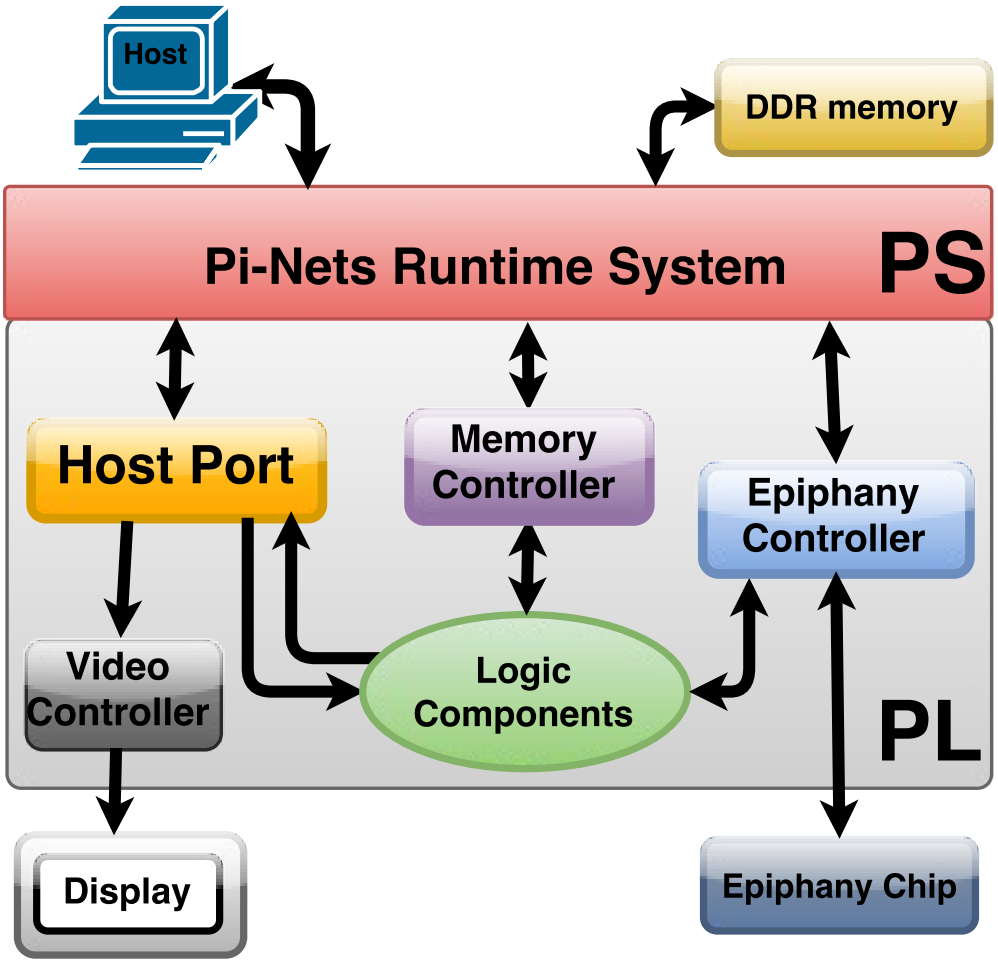

Figure 4.1: Operating infrastructure diagram

4.2 Host Port interface

The Host port is a module in the PL whose function is to transmit test data from the host computer to the other components in the PL. It's called Host port because it's a module on the boundary of the PL who links many components together to transfer data between the host PC and the connected components. The Host port distributes the data to other components using a 3 bit ring bus, that in 7 clock cycles delivers 21bit of data, being able to communicate with all the components that are linked to it. A simple representation of the Host port connections in the PL are represented in Fig. 4.2. The ring bus function is to shift the data, at each clock cycle, through the components, and if the data arrives at its destination, the respective components will accept it. The components can also produce an answer by writing the data in the bus ring, and as it arrives the Host port the data is forwarded to the host PC. It is also possible to attach Peripherals to the Host Port, as input or output sources, that can also be addressed by the components.

The main modules of the host port will be further discussed.

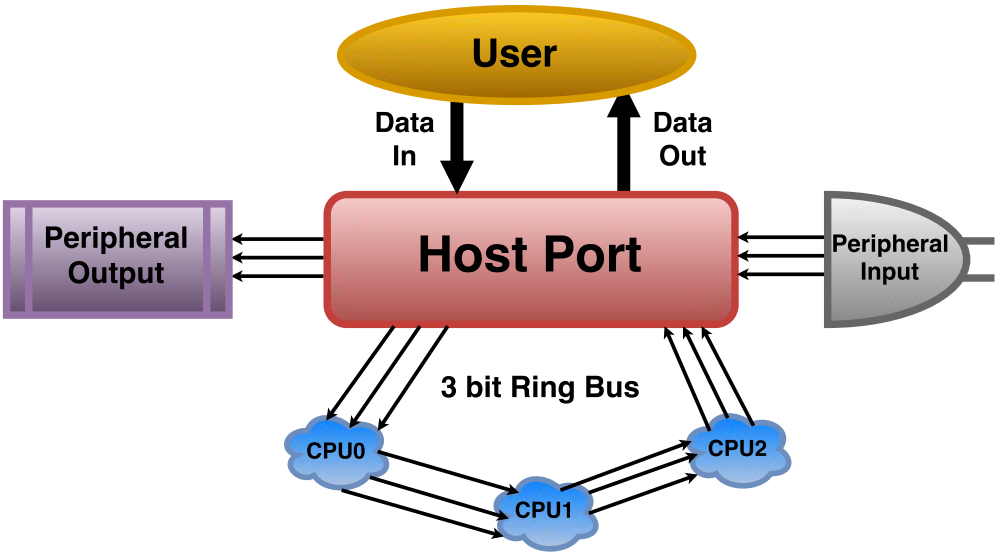

Figure 4.2: Host Port interface connections

4.2.1 Host Port User input interface

To link the host PC with the Host Port module different solutions can be implemented. Two possible solutions will be further analysed.

User input through UART

The first possible implementation would be using a UART. In this way the module would communicate with the user by typing commands on the UART and also display the output on the users's UART interface terminal. The UART communicates in a serial manner and delivers 8bit at a time. A UART implementation is also easily obtained from other projects and allows to be linked to the project without much difficulty. But using a UART configuration would be difficult to communicate to multiple systems that are working in parallel, considering the ER4.

User input through Ethernet

The second option would be to use the Ethernet interface. With this interface, Ethernet packets could be intercepted by the Zynq, containing each packet a big amount of data. With a proper configuration the data from the packets can be forwarded to the host port configured on the PL. The Ethernet interface on the Zynq is linked to the PS, it's possible to link it to the PL using the EMIO. In alternative, a Software application on the PS could forward the packets to the PL. Also, in case the host port wants to answer over the Ethernet interface, a Software application could be implemented on the PS, to create the Ethernet frames.

Communicating through a Ethernet interface allows to manage multiple systems working in parallel, as they are connected in a network.

This interface was chosen, being the the most suitable for this implementation, in which the Ethernet interface is linked to the PS, meaning that a software application is necessary.

4.2.2 Host Port interconnections on the PL

The Host Port block is the link between the host PC, the components on the PL and some external peripherals.

A similar interface was designed for the Spartan-6 FPGA in another project of the institute, also related to a similar project as the ER-4, and most of its logic can be reused in this architecture. This block provides simple IO signals for user input and handles the ring bus interface, as well as the peripheral's input and output. The external peripherals have the same bus interface type as the ring bus, where the only difference is that they are not connected to the ring bus, but to a separate bus. The Host Port is so divided in to two main interfaces, the user data interface and the interface to the components.

The interface to the components will be considered as a black box, as this logic is used as it is, providing the proper interface for the ring bus and for the external peripherals. The black box has four simple user IO signals that are used in the data transfer to the host PC. The read and write are independent signals, having each one his own data port of 21 bits and a enable signal. This black box will be linked to a module who will connect to the host PC interface.

Considering that the Ethernet interface is to be used and that the data is transported through the PS, a communication port between the PS and the PL has to be configured.

To connect the host port to the PS, one of the GP ports of the Zynq can be used. The communication protocol between the PS and the PL is the AXI protocol. To forward incoming Ethernet packets to the host port, at any moment of reception, the PS should be configured as master and the host port on the PL as slave.

In this configuration, the transactions are always controlled by the PS, allowing it to write data on the host port at any moment.

To allow the host port to send data to the host PC, when data is made available, a PS interrupt has to be configured, in a way that when the host port wants to send data from the PL to the host PC, an interrupt to the PS is enabled, forcing the PS to read the Host Port's address and processing the data to the host PC.

This logic block has to be designed with regard on the black box block that will handle the logic of the ring bus and the external peripherals. A representation of this proposed design is shown in Figure 4.3.

4.2.3 Host Port Software

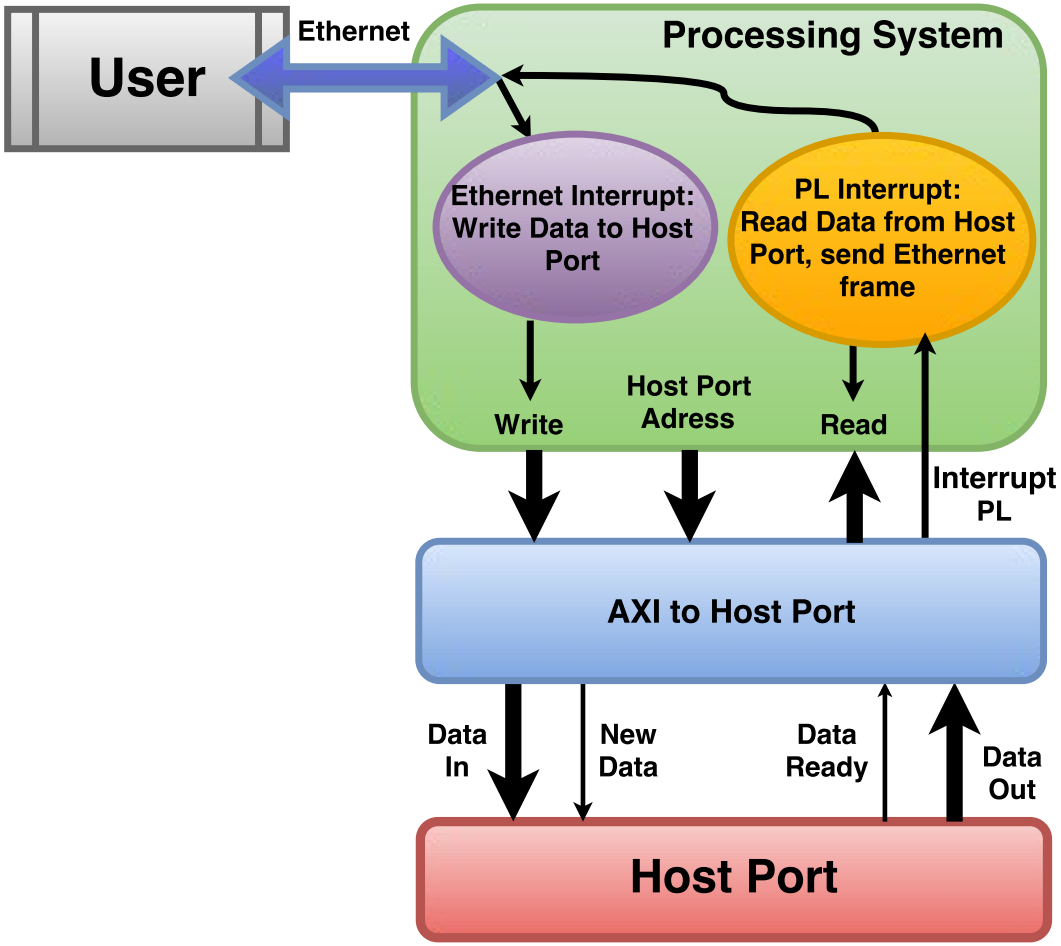

Figure 4.3: Host Port diagram

To transport the data from the Ethernet interface to the PL, a software application has to be executed on the PS. In the PS's point of view, the host port can be accessed by writing in to the mapped memory space. When the Host Port wants to send data back to the host PC, some software code has to be executed to read the data from the host port, to prepare a Ethernet frame and to dispatch the frame.

As a Software base for the Zynq, multiple solutions are available. Multiple Linux distributions are available for the Zynq[25], but also custom standalone software can be implemented using the available runtime system.

Ethernet using Linux as OS

Using a Linux distribution, the Ethernet interface could be accessed via software. An application could be designed to analyse the incoming Ethernet packets and to filter the ones with destination to the Host port.

To read data from the Host port, on the PL, a Software interrupt could be configured. That would execute an application that would read the data and pack it in to a Ethernet frame to send it back to the host PC.

By using a Linux distribution one has a full Operating System support, where every external hardware device can be accessed, multiple configurations can be performed and a full application execution environment is granted. On the other hand, it also requires some processing abilities from the processing system, as it gets busy with the OS tasks, but some of these tasks are not particularly relevant for the implementation.

Having a full OS on the system with all its features would be very comfortable, even with the extra loss of throughput. But there is an inconvenient, all the features that were implemented on the runtime system wouldn't be of much use, because the developed program was thought to run standalone and not on top of the Linux distribution.

Ethernet on the Runtime System

Using the Ethernet interface combined with the runtime system provides another option, one could use the functionality of the runtime system, who already implements a Ethernet initialisation. Also the Ethernet packets with destination to the Host Port were a planned entry on the runtime system. The PiNets Ethernet protocol defines a Host port message number. This number is analysed by the runtime system when a Ethernet packet is received. The treatment of the packets for the Host port was not defined yet, but the entry is ready to be integrated and could so be linked to an interrupt handling routine that would forward the incoming data to the host port.

4.3 Display Text Output

Another service that the infrastructure has to provide is an interface to an external Screen to display text.

This interface is to be attached to the Host Port peripheral input, where 21 bits of data are transferred in a 3 bits bus, over 7 clock cycles. Just the least significant 17 bits have valid data for the peripheral.

The components in the PL access the peripheral interface by using the ring bus, setting the attached peripheral as destination for the transferred data.

The physical interfaces in the ZedBoard that can perform such task are the VGA interface and the HDMI interface. The Parallella board just has HDMI output, but both video interfaces will be further discussed.

A project that implements a VGA text output has been developed with another hardware platform. This project was developed by the author of this thesis in the context of a Project Work and its logic will be considered to be reused in this implementation. The specification of this project will be given in more detail below.

4.3.1 HDMI

The HDMI interface offers a high resolution capable video transmission. One can produce high resolution images on the Screen using this protocol. It also offers the possibility of transporting, in the same interface, one or more audio channels.

The objective of the implementation is to display text and the high resolutions does not bring any advantage to display simple text. The possibility of adding a audio channel to the implementation could be more attractive, but to implement such a feature additional time needs to be spent in the design of such component. The entire architecture of the audio data transport would have to be designed. For the goals of the operating infrastructure, the audio feature does not represent a fundamental need.

If at the end of the implementation of all fundamental blocks some time remains, then the design of a audio interface could be carried out.

4.3.2 VGA

The other available video interface is the VGA interface. This interface is not as good as HDMI, regarding its resolution and quality, because VGA uses a Analogue Circuitry and the HDMI is completely digital. Some image quality losses, with the VGA interface, can happen. The VGA interface is capable to be configured with multiple resolutions, at which, the maximum resolution can be considered too high for the use of this implementation.

By using the VGA interface, a project that was implemented on another hardware board could be reused in this implementation. It configures a hardware block capable of managing different sections of the screen, allowing the applications to write in different sections of the same screen. This project could be interesting to integrate, such that the different components on the PL could have singular access to different sections of the screen, for displaying text purposes. This implementation will be further presented in the next subsection, and its specification and capabilities will be explained based on [26].

4.3.3 VGA text output Specifications

This VGA implementation generates text output in a VGA monitor, in a way that different applications can conveniently interface a text output on the FPGA. It integrates the VGA variants of 800x600 and 1024x768 pixels, with output rates between 40 and 100 MHz and frame rates of 60 to 75 Hz. With these resolutions text pages of 100x38 or 128x48 are generated, with 8x16 pixels for each character.

The VGA block has a text page buffer that is read periodically for image output and is written with a 16 bits write port.

The main interface signals of the VGA Circuit are:

CLK System clock (100MHz)

DIN 16-bit-Inputport

DSTR Write enable signal for Input

R,G,B,H,V Video data inc. Synchronisation Signals

The writing position is managed for 8 output windows, to provide independent output from multiple application circuits.

A RAM memory with the character pattern generator is integrated in the project. The character generator provides the pixel position of the characters to be written in the VGA window.

The inputs to DIN are done in the following formats:

www|111|aaa|ccccccc Gives character code c out on writing position of window w (a=0).

www|yyyyyy|xxxxxxx Being y the vertical and x the horizontal coordinates, set the position for window w, with y <52.

Visible sign expenditure move the cursor position to the right, special ASCII characters can also be used, they are represented in Table 4.1.

Control functions, such as the display of the writing position, setting the foreground and background colours or the position of a window can be done with the a-field, respecting the following format:

Decimal	Hexadecimal	Abbreviation	Name
0	0x00	Nul	Null character
8	0x08	BS	Backspace
9	0x09	HT	Horizontal Tab
10	0x0A	LF	Line Feed
11	0x0B	VT	Vertical Tab
13	0x0D	CR	Carriage return

Table 4.1: Implemented ASCII control codes

- If 000 then it will send a new char to the selected window.
- If 001 it sends the new foreground colour contained in **ccccccc**
- If 010 it sends the new background colour contained in **ccccccc**
- If 011 it enables a blinking cursor in the actual writing position, send 1 to enable and 0 to disable in bit 0, by default it is disabled
- If 100 then it will command to delete the whole selected window

With this implementation it is also possible to configure an automatic start up output.

4.3.4 ZedBoard VGA output modifications

To use the presented VGA project in this implementation, some adaptations have to be made.

The project was implemented on an hardware platform, that had a 2 bits width DAC for each colour signal. The ZedBoard on the other hand has a 4 bits width DAC, enabling a wider colour range. But the output signals of the implementation have to be adapted to the 4 bits width DAC from the ZedBoard. Also the command who changes the foreground and background colours has to suffer some adaptation.

4.3.5 Host Port to input modifications

The VGA block's data input is 16 bits width and assisted by an enable signal. Considering the peripheral output from the Host Port, that sends 21 bits of data in 7 clock cycles in a 3 bits width bus, a module has to be integrated to translate the transmitted data to the VGA block's input.

This module would contain some registers to store the incoming data from the Host Port and as soon as all the bits were received, the valid 16 bits would be sent to the VGA block's input port.

4.4 PC Host Program

To externally control the interactions on the Zynq, a PC host program has to be designed. This Program communicates through the Ethernet interface with the Runtime System, using the PiNets Protocol.

The basic functions of this Program are to create and send Ethernet packets, containing the different PiNets messages types and also to receive the Ethernet Packets that are sent from the Zynq.

The main services that the program offers are:

- To check if the Zynq is Online.
- Download PiNets Applications to the SD card.
- Configure the FPGA and start a PiNets Application.
- Write status data to RAM.
- Write and receive Data from Host Port.

This Program should be supported for Windows and Linux, or, at least, with very few modifications be portable to the other platform. As programming language, C will be used to write this Program. It should be executed on a simple command window and display a General User Interface (GUI) with the available options.

The details of these Program's capabilities and its Ethernet configurations will be further discussed.

4.4.1 Ethernet Configuration

The Ethernet interface is the main hardware component that the program will use, and, considering the cross-platform compatibility, it will also be the biggest problem. To overcome this issue, a convenient Ethernet library has to be chosen, that is compatible with both OS's and provides the relevant Ethernet functions. The reasonable solution found is to use the LibPcap library on Linux, from [27], and the WinPcap Library on Windows, from [28]. They have similar functions and just small changes.

The Ethernet packets will be received, sent and handled as raw Ethernet. The Ethernet communication interface on the PiNets Runtime system was implemented using Raw Ethernet. Another Ethernet transport layer could be adopted, but some modifications would have to be performed on the PiNets Runtime System.

Prior to the start of communication the program must initialise the Ethernet interface on the PC, using the functions of the referred library.

As to receive the packets coming from the Zynq, an Ethernet packets filter is configured. This filter should filter all Ethernet frames that came from the Zynq.

4.4.2 User interface

By executing the program on a Command window a simple user interface should be displayed, so that the user can select one of multiple options. For convenience of the user, all the available PiNets messages that can be constructed are displayed on the Terminal. A way to access them must also be designed, in a way that the user can select one of them at the time, and afterwards select and construct another message.

A simple menu with numbered options would be a possible solution. An interface were the user has to input some commands could also be implemented.

4.4.3 Runtime System functions

The Runtime system provides some features that will be used by this Program, as it handles the received Ethernet packets containing Pi-Nets Messages. To test the functionality of the runtime system some functions were constructed by the developer of the runtime system. The input of these functions is static, as they were designed to test the system, but one could

modify it by filling it with user input. These functions need to be integrated and adapted in to the program, as they provide access to some of the key features of the runtime system.

The implemented functions will shortly be described in the following section.

Check if Zynq is Online

To check the availability of the Zynq one can send a message to it that will force a simple answer. The Zynq will answer with a Ethernet frame containing its configured name.

Download PiNets Applications

To transfer PiNets Applications to the Zynq's SD card, this function is used. It transfers the three files, who constitute a Pi-Nets application, to the Zynq, by sending multiple Ethernet frames.

To use this function in the Program, some method to load the files has to be provided. The standard file access functions can be used to open the files and afterwards the Ethernet frames can be prepared for downloading. User input could be given to the File source tree to indicate which files to send.

Start PiNets Applications

With this message type one can start PiNets applications on the Zynq. The user must enter the corresponding information of the application to be started.

Write status data to RAM

By using this function it's possible to write directly in to the DRAM. The user has to provide as input the Addresses and the Data to be stored.

4.4.4 Write and receive Data from Host Port

The Host Port feature was implemented on top of the runtime system, but its Ethernet message handling was already accounted. The function that handles the received data with destination to the Host Port had to be designed. Considering that design, a function that handles host port IOs has to be integrated in the program.

The user will have to type in the data that is to be sent to the Host Port and the Ethernet frame is constructed based on the host port PiNets Protocol.

The data that the Host Port sends through the Ethernet interface back to the host PC must also be accounted. By implementing a listening mode, one could capture these packets, where the program is waiting for incoming packets.

4.5 Memory Interface

To give the system memory access to the components configured in the PL, a module with a convenient interface has to be designed.

The DDR RAM chip is placed externally, regarding the Zynq chip, and has shared access. It can be accessed by the PL and by the PS's DMA, being, as well, directly linked to the L2 cache. The prioritisation of the different inputs is internally managed by the Zynq, it implements some logic to avoid data starvation and keeping a fair shared access.

The Zynq offers four High Performance Ports in the PL, that give access to the DDR memory interface and to the On-Chip Memory (OCM). Each of them can be configured with 32 or 64 bits as data input. The Address range can be configured using the Xilinx tools. To communicate through these High Performance ports the AXI Protocol must be used.

A representation of the architecture of such memory interface is represented in Figure 4.4. Each of the represented blocks and its communication paths will be discussed in the following subsections.

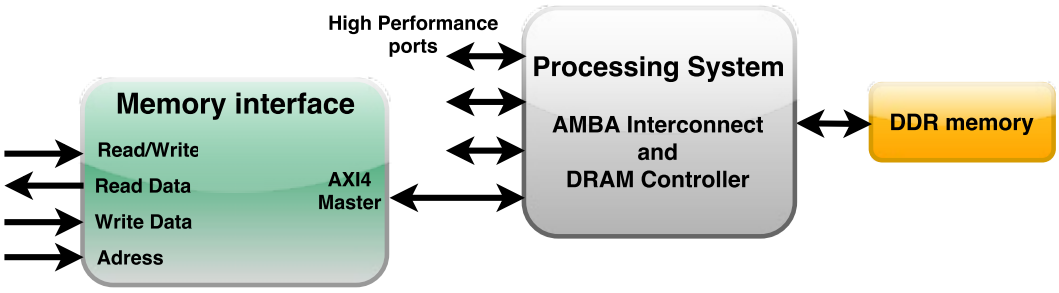

Figure 4.4: Interfacing the DDR memory, on the Zynq

4.5.1 Using the High Performance ports on the Zynq

To interface the Zynq's High Performance Port and access the DRAM, a communication logic using the AXI protocol has to be designed, where the module in the PL is the Master and commands the transactions.

To design an AXI interface one must first choose between the different AXI protocols. The available protocols are AXI4, AXI4-Lite and AXI4-Stream. The objective is to have the highest throughput possible without having any loss of data. So, the AXI-Lite protocol is immediately excluded as it serves for low throughput access. Also the AXI-Stream can be excluded as this protocol has a high throughput but is adapted to transport video data, complicating the use for single accesses. The most suitable AXI protocol for this application is the AXI4.

Using the Xilinx tools its possible to create an example block that uses the AXI4 protocol as communication protocol, this block can be used as starting point for the implementation of a AXI4 capable block.

4.5.2 AXI4 Master configuration

The AXI4 interface example implements a memory test module, using the AXI4 protocol in Master configuration. A pre-configured memory range is written with some data and afterwards the same address ranges is read and verified if the read data matches the written one. By analysing this example, some logic about how to implement the AXI interface could be reused. For example, the implementation of a state machine to initiate a write or read transaction and the address and data channels to initiate the protocol.

The AXI4 protocol is also capable of bursts. The Burst can be configured in many ways, the protocol defines Burst sizes of up to 256 beats. Considering the configured burst size, the data Words will be sent successively at each clock cycle. If a burst is configured, only one Address is sent to the interface. The Address is internally incremented, where different methods of increments can be configured, depending on the memory organisation and type.

4.5.3 Input signals

To give access to the components configured in the PL to the memory interface, some simple interface signals have to be designed. The interface should be capable of initiating a read or write command as an input signal is received. Some signals to provide the address and the data words must be included as well. The bus caring the read and write data have its own data bus, in a way that no bidirectional bus has to be created.

The read data is signalised by a second signal that goes high when the data is asserted on the bus.

For the Burst option, another signal should be configured, that would signalise different sizes of bursts for different applications situations that demand a burst.

4.6 Epiphany Integration

The Epiphany co-processor present on the Parallella board is to be integrated in the project's infrastructure. The Epiphany chip is physically linked to the FPGA IO's on the board and its interface communicates with the eLink protocol. To make the Epiphany accessible, the hardware that configures the eLink protocol has to be integrated on the FPGA. The eLink hardware is also linked to the DDR memory port by one of the Zynq's HP ports, as shown in Figure 4.5.

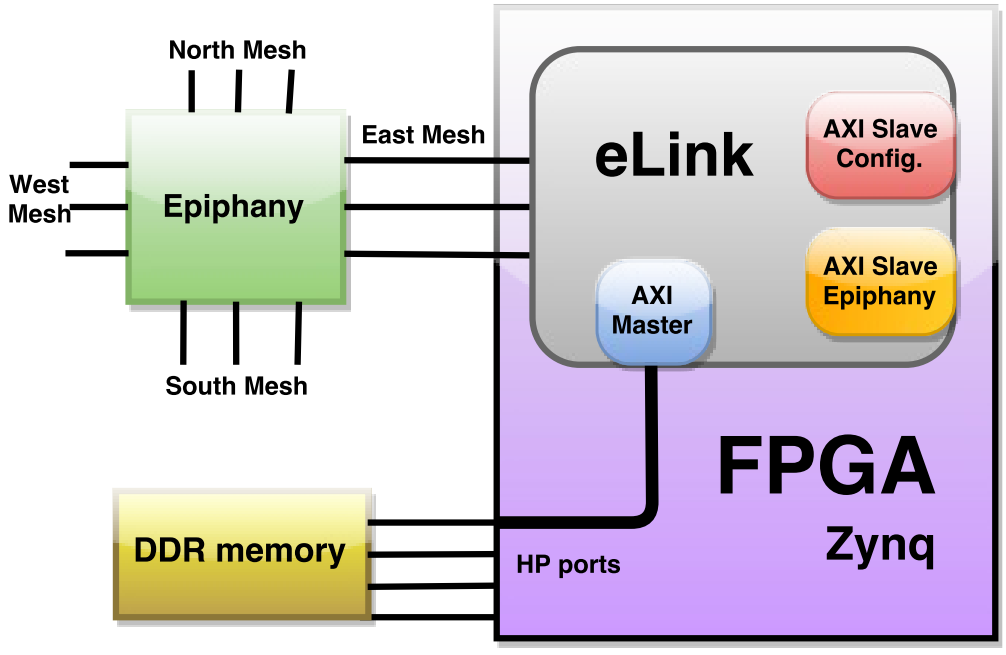

Figure 4.5: Epiphany connected to the FPGA through eLink and to DDR memory

The Parallella project is as much as possible open source, and all its source code files can be downloaded directly from its repository on GitHub [29]. The hardware project of the eLink protocol that communicates with the Epiphany chip can be found on this repository and it can be reused in this project, to integrate the Epiphany.

The Epiphany is, in this implementation, linked to the PS through AXI interfaces, one for the actual Epiphany chip and the other for some configuration registers present in the eLink hardware.

The Epiphany chip could be connected simultaneously to the PS and to some other hardware components configured on the PL. That could be achieved by attaching at the AXI interface an AXI interconnect, that would multiplex the access of both interfaces, like in [30]. The concept applied to this implementation is demonstrated in Figure 4.6.

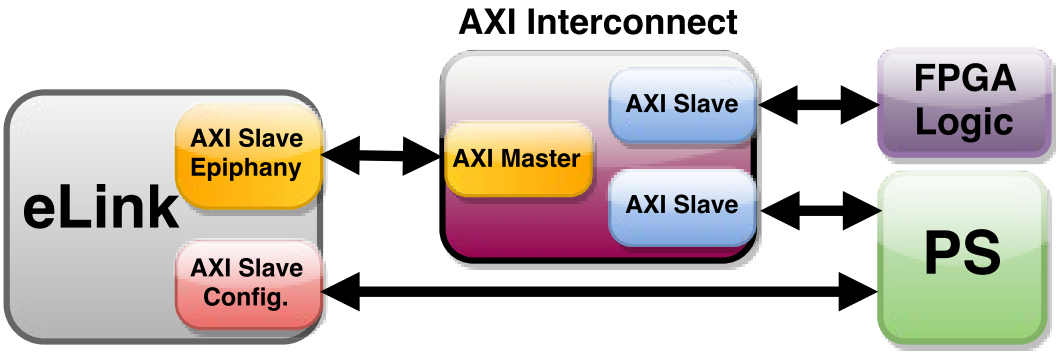

Figure 4.6: Epiphany double Master access through AXI interconnect

To be able to use the Epiphany's co-processor capacities one has first to configure it. That could be accomplished by loading some instruction code to the chip's internal memory and set the start of its execution. The instruction code could be loaded by the PS or from some PL logic. As this can be considered an initialisation task, the piNets Runtime System could also be upgraded to initialise the Epiphany Chip with some instruction code.

The Epiphany chip has its own instruction set, which is consisted of both 16 and 32 bit width instructions. The full instruction set can be found on [6].

To support this instruction set and the compilation of programs capable of being executed on the Epiphany chip, has the Epiphany chip Manufacturer Adapteva created some support tools. They consist of a specifically designed SDK for the Epiphany chip. The support package includes a vast repository of routines that can be executed on the Epiphany chip and also routines that control the program execution. As the Parallella board was thought to be used with a Linux distribution running on the ARM, the routines in the Parallella repository are designed to be executed on an Linux OS. To make the routines available on an standalone system they would have to be rewritten.

Some concepts about the memory organisation and other initialisation aspects of the Epiphany chip will be discussed in the next sections.

4.6.1 Epiphany's Memory Architecture

The Epiphany has a address space consisting of 2^{32} bytes in which four bytes are aggregated to view the memory with a 2^{30} width, consisting of 32-bit words.

The Epiphany memory architecture has a byte order configuration as little-endian.

Each core node has a local memory range that is accessible by the core node itself. The address range starts at 0x0 and ends at the address 0x00007FFF.

Each core node is globally addressable, from chip external components or from other core nodes. To address each core node, a unique ID is associated with each core. This ID is identified by a total of 12 bit, situated in the most significant bits of the address space, where the upper 6 bit indicate the core row and the other 6 bit the core column. A complete representation of the memory space, in terms of an individual core node and all of them together, is represented in Figure 4.7.

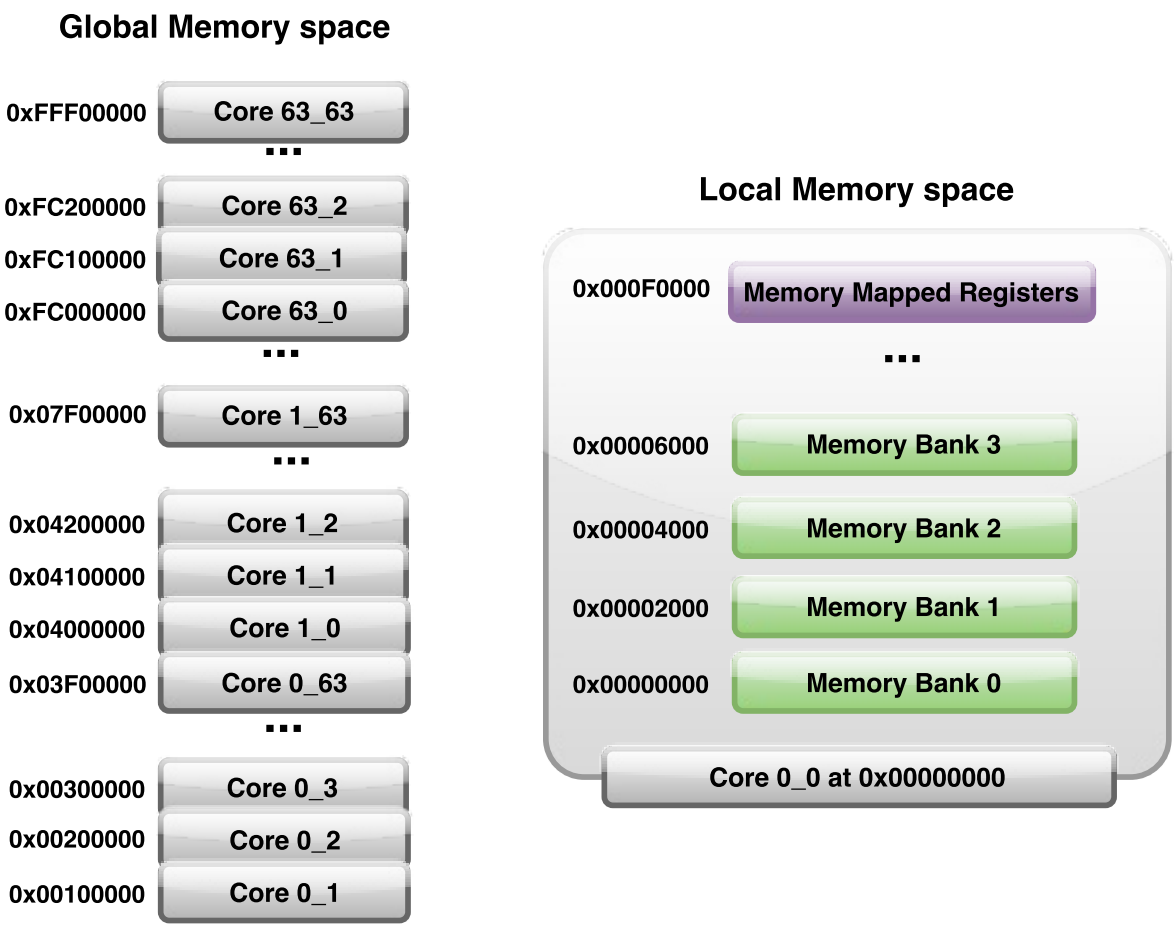

Figure 4.7: Epiphany memory organization

Instruction code and other data can be stored in the local memory and also in external memory. To achieve the best performance, the instruction code and data should be stored in different local memory banks.

The CPU registers are also memory mapped, together with other configuration registers, they are mapped above the 0x7000 local address on all cores.

4.6.2 Epiphany's eMesh Network

The Epiphany chip has a special network topology called eMesh. It is capable of interconnecting all the core nodes of the chip and has four separate routing links to the exterior. The special node linking network of the Epiphany allows to attach multiple Epiphany chips and aggregate them together in a single network.

The memory mapping of the Epiphany chip supports up to 4096 node cores in single shared memory system. In a practical implementation one would probably map other external peripherals, like DDR and other IOs. But, even considering external peripherals mapping, it is possible to configure multiple Epiphany chips in a single shared system, thanks to its network topology.

Each node is only connected directly to the nearest node. Every route in the network is connected to the north, east, west, south and to a mesh node. Where the terms north, east, west and south define the continuation of the network if multiple chips would be integrated

in the same system. All four interfaces can be connected to other external peripherals and alternatively being left unconnected.

Transactions move through the network until they reach the pretended destination, having each routing hop a latency of 1.5 clock cycles. In an 16 core chip, the number of clock cycles to go from the top to the bottom would be 6 clock cycles.

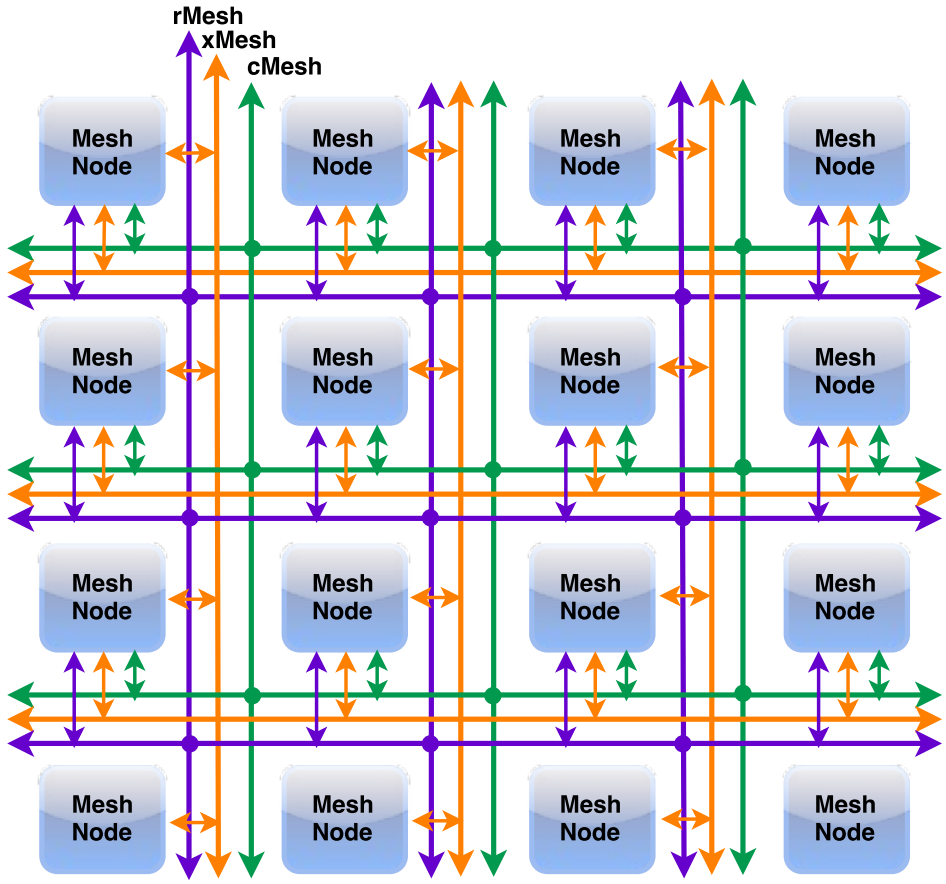

Figure 4.8: Epiphany eMesh, adapted from [6]

The eMesh Network is constituted by three separate bus structures, as represented in Figure 4.8. They serve for the needs of different transaction types. They are:

- cMesh: This network is used for write transactions, with destination to an on-chip core node. This network connects a node to all four of its neighbours.
- rMesh: To process read requests is used the rMesh network. Similar to the cMesh this network connects a node to the four neighbours as well.
- xMesh: With this network it is possible to pass write transactions with destination to the chip's external resources. The xMesh network is split in to north-south and west-east. This network is responsible to connect other Epiphany chips in a single system structure.

4.6.3 Epiphany initialisation and Control

To initialise the Epiphany some internal registers have to be configured. The implementation of the eLink protocol converts AXI inputs from the PS in to Epiphany output signals, in the

PL. To this hardware set-up, some control registers are also added, and, when starting up the system and programming the FPGA, a reset signal will initialise all these registers with zeros. A initialisation has to be done to properly configure the Epiphany chip.

At the initialisation, the memory of the Epiphany is filled with random numbers, as well as the interrupt vector table. These values can be rewritten to zero, or some other handling is required.

After the initialisation procedure, instruction code can be downloaded to the Epiphany's on-chip memory. Multiple configuration schemes can be used, depending on how the Epiphany is to be configured. To enhance the throughput of the Epiphany chip, one can download instruction code to each of the core nodes and execute it in parallel.

To initialise code execution on a specific core of the Epiphany, one can write in a specific core register, who will point to the start address of the instruction code.

4.6.4 Pi-Nets Runtime System on the Parallella Board

To integrate the Epiphany co-processor in the infrastructure designed on the Zedboard, it will be necessary to port the created system to the Parallella board. The Parallella board has a different hardware layout than the ZedBoard. The peripherals are connected to different IO pins. Also the Zynq chip on the Parallella board, that is used on this project, is slightly different than the one present on the ZedBoard. The Parallella uses a Zynq-7010 and the ZedBoard uses a Zynq-7020.

The hardware configurations of the designed project will have to be adapted for the new hardware platform. As soon as these configurations are set up, a new BSP can be created, which will have all the necessary support tools to configure the piNets Runtime System as a standalone application on the Parallella's Zynq. Some upgrades to the Pi-Nets runtime system have to be done to integrate and initialise the Epiphany chip.

Chapter 5

Implementation

In this chapter the implemented Project will be presented and all its functionality will be explained, the Host port, the VGA output, the Desktop PC Application, the memory controller, the Epiphany integration, as well as the generated system will be further discussed.

5.1 Host Port

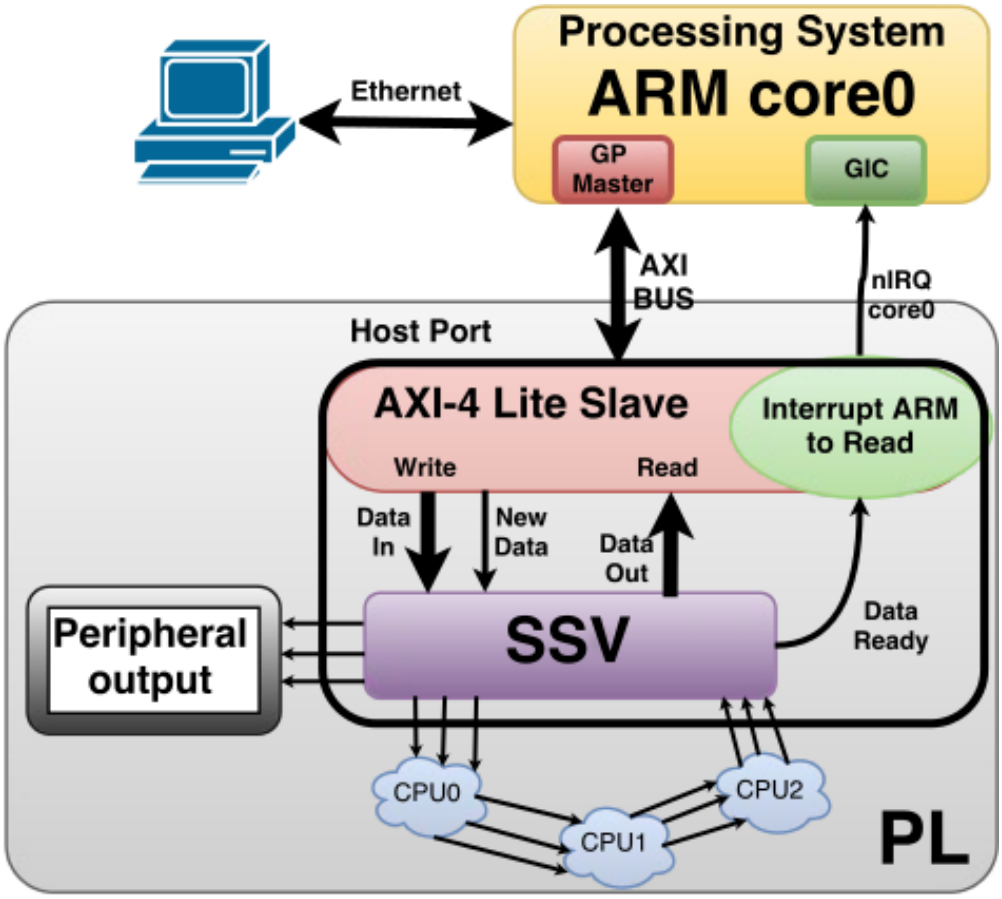

Figure 5.1: Host Port Diagram

The Host Port is an important block of the infrastructure constituted by multiple components. These components are interconnected to form a communication bus, that goes from the Desktop PC to the Host Port in the PL, passing through the PS. The components that constitute the Host Port are represented in Figure 5.1, they will be briefly described.

To communicate from the PL's Host Port, through the runtime system running in the PS, a Software solution to transport the data had to be created. Which has the capacity to receive data from the Ethernet interface and forward it to the Host Port, and to receive data from the Host Port and forward it to the host PC, through the Ethernet interface.

When the host PC wants to send data to the Host Port, an Ethernet frame is transmitted to the PS. Then, some Software routine is executed in the PS to handle the data of the received frame. This data is sent to the Host Port in the PL, through an AXI bus. The data is received in the PL, at the host port, and processed by the System SerVer (SSV) entity (handled as black box), who connects to the ring bus.

In the inverse case, where the data is transmitted to the host PC, the host port starts with triggering a signal that interrupts the PS, who processes an interrupt handling routine that will read the address of the Host Port through an AXI bus. In the same routine an Ethernet frame is constructed and sent through the Ethernet interface.

In the next subsections the details of each of the linked components will be more extensively discussed. This section will cover the Host Port implemented in the PL and its AXI protocol, the code that is processed in the ARM to handle the Host Port data, the configured interrupt and the Ethernet frames that are processed.

5.1.1 Host Port in the PL

The Host Port in the PL is constituted by two main blocks. One block is the SSV who connects to the ring bus and also to other peripherals, the other main block is responsible for the data transaction between the PL and the PS, using an AXI protocol.

These block's functions and signals are further explained in the following subsection.

System SerVer (SSV) block

The Host Port is a slow circuit, its ring bus is only 3 bits width. It receives 21 bits of data from the PS and these bits will be sent through the 3 bits width ring bus in 7 clock cycles, as represented in Figure 5.2.

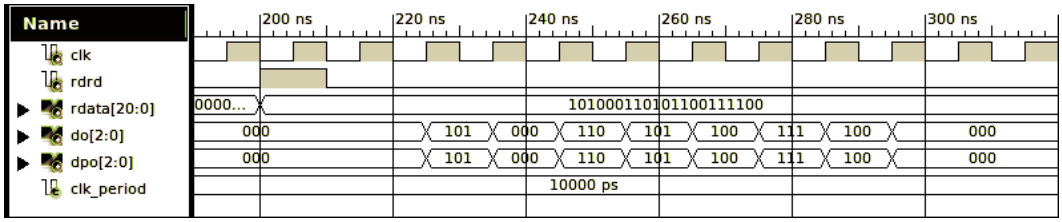

Figure 5.2: Timing diagram of the SSV block

The SSV block presented in Fig.5.3 is responsible to flow the data through the ring bus. This ring bus shifts the data at each clock cycle to the attached components. In idle state the bus will have all the bits with the value 0. This block was adopted as it is and attached to the AXI Slave interface with the custom output signals, the VGA peripheral was also attached to its respective interface. The signals that were used from this block and linked to the AXI Slave block were:

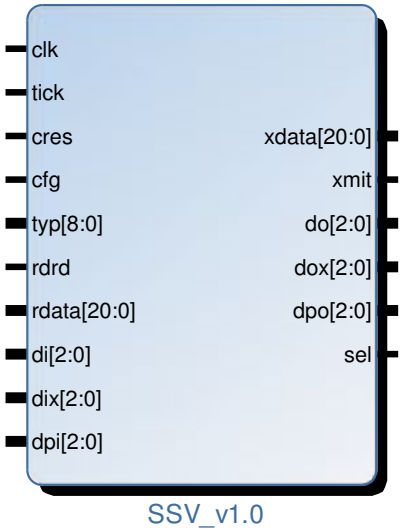

Figure 5.3: SSV block

xdata as the SSV block's data output.

xmit to signal that valid data is in the xdata bus.

rdata as the SSV block's data input.

rdrd to signal that the input data is valid.

The output peripherals are attached to the *dpo*, this signal transmits 21 bits of data over the 3 bits width bus in seven clock cycles. When received data is decoded to the output peripheral, the data is forwarded to *dpo*.

It is also possible to input data from a peripheral using the signal *dpi* with similar functionality as the *dpo*, but regarding that the transaction has the inverse direction.

The ring bus is linked through the signals *din* as input and *dout* as output. As there weren't any components available to link to the debug ring, the two signals were linked together to have a feedback of the sent data, as represented in Fig.5.5.

AXI Slave block

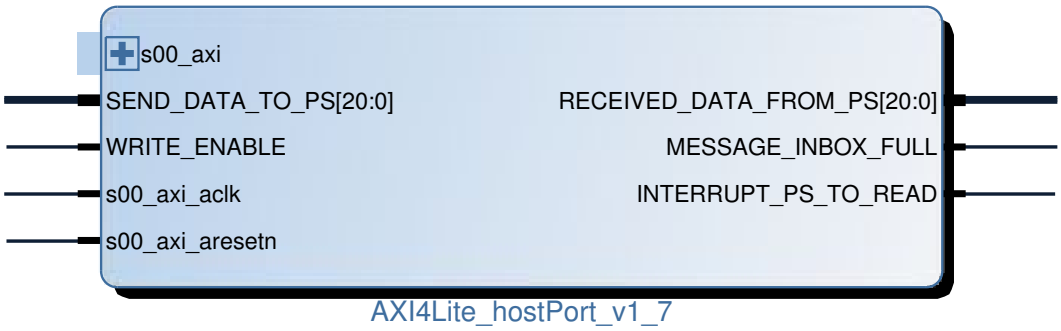

Figure 5.4: AXI Slave block

The simplest and also slowest AXI protocol to transfer data between the PS and the PL is the AXI4-Lite protocol. Using this protocol, just one data transaction, between PS and PL, can be validated at a time, this protocol is also not burst capable. For the needs of the Host Port the AXI4-Lite protocol was considered to be sufficient, as the Host Port block in the PL has a slow behaviour.

The Zynq ports that link the PS to the PL have different characteristics. For this implementation the most suited ports are the General Purpose ports, as they can be used in a configuration where the PS is Master. They are appropriated for any general use within PS and PL communication, and in this Master configuration they can be linked to multiple slaves in a single GP port.

The AXI Slave block is mapped to the PS with an address range of 4Kbytes, configured with the minimal value. For the AXI Slave block, one address would be enough, but the minimal value has to be selected. The Offset Address was automatically configured to 0x43C00000. The implementation does not verify the received address from the PS, so, for this purpose, an AXI Interconnect from the Xilinx IP Cores is connected in between, linking the data to the right slave. It is possible to write or read from the PS in the entire configured address range, obtaining the same result, as the entire range is configured to behave like one single address.

The AXI protocol assures that the data received from the PS is valid, using the handshake scheme between slave and master. The aim of this block is to translate the AXI protocol signals in to the input signals of the SSV block. For this purpose some simple signals were created, as represented in Fig.5.4, where the AXI bus is converted in to custom build signals. This block is able to signal the arrival of new data and to forward received data to the PS. To receive data from the PS, the signals MESSAGE_INBOX_FULL and RECEIVED_DATA_FROM_PS were created, where the signal MESSAGE_INBOX_FULL is set high for one clock cycle whenever new data has arrived. To forward data to the PS, the signals SEND_DATA_TO_PS and WRITE_ENABLE were created. The signal WRITE_ENABLE has to go high for one clock cycle to validate the data and interrupt the PS to read it.

This block was designed to interrupt the PS when data is received from the SSV block, forcing the PS to read the host port's address. The interrupt signal linked to the PL is the PS's core 0 private interrupt.

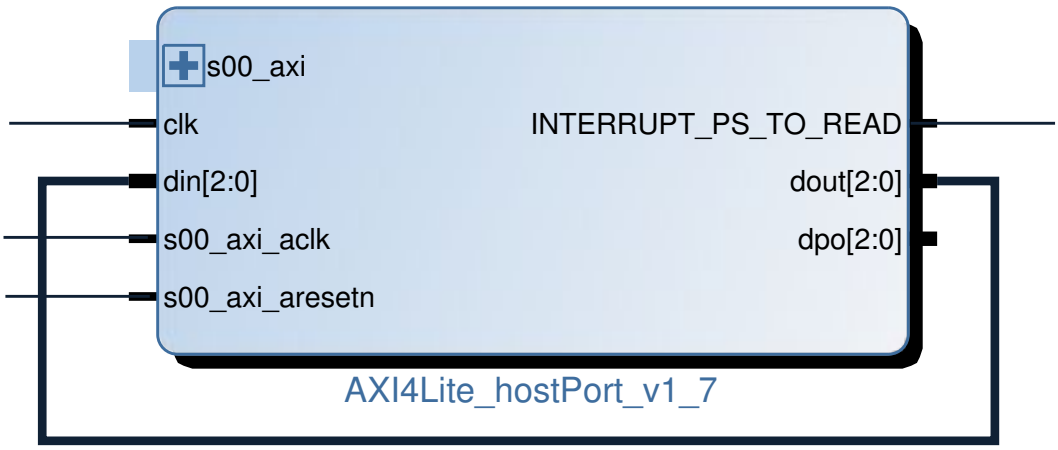

Figure 5.5: Host Port block and debug ring bus

The block represented in Fig.5.5 is the final implementation of the Host Port block in the

PL, where the SSV block and the AXI Slave block are joined together.

5.1.2 Host Port functions in the PS

The PS is the communication link between the Host Port and the Ethernet interface, and it manages the transactions in both directions.

Essentially, there are two functions in the PS who deal with this communication, one of them to read the data from the Host Port and forward it through the Ethernet interface and the other function forwards the Data that is received from the Ethernet interface to the Host Port.

Each function is called when an interrupt source is triggered. These functions will be further discussed in the next section.

Ethernet to Host Port

When an incoming Ethernet frame interrupts the PS, its content is analysed by the runtime system to identify the message type. If the message type is decoded as the Host port, a function is called to write the data to the Host port.

The data contained in the Ethernet frame is arranged in bytes. It has to be converted in a 32 bits variable, to align the data for the Host port. To send the 32 bits of data to the Host Port, the PS just has to write the Host port's address with the data. The Host Port just receives the least significant 21 bits of the data, the rest is discarded. The AXI interconnect routes the data directly to the intended slave in the PL.

Host Port to Ethernet

In the case where the Host Port wants to send data to the Ethernet interface, an interrupt will trigger the PS to run an interrupt handling routine. In this routine the PS reads the Host Port address containing the 21 bits of data and saves it in a 32 bits variable. Then, an Ethernet frame is prepared and its header is written, calling the appropriated functions from the runtime system. The 21 bits of data are copied in to the prepared Ethernet frame, and, at last, a function is called to send the data through the Ethernet interface.

Interrupt IRQ core 0

This interrupt ensures that the interrupt is processed by the core 0, as the PS is running in bare metal and the core 1 does not have the pointer to the code that is processed by the interrupt handler. Also the configurations of the Ethernet interface are performed on the core 0, allowing to forward the data directly through the Ethernet interface, using the core 0.

The interrupt is initialised at the start of the runtime system, where its Interrupt Service Routine (ISR) is linked and the interrupts are enabled.

5.1.3 Ethernet frame

The Ethernet frame is constructed based on the PiNets Ethernet protocol, in which the Host Port code is the number 7.

The raw Ethernet frame that is sent from the PS to the Ethernet interface is represented in Table 5.1.

destination MAC	Zynq MAC	0x6712	21 bit Data

Table 5.1: Ethernet frame sent from the PS

The Ethernet frame that is received by the PS and forwarded to the Host port has the structure shown in Table 5.2.

Zynq MAC	Source MAC	0x6712	Zynq MAC	0x12B9B0A1	0x7	21bit Data

Table 5.2: Ethernet frame that the PS receives

5.2 Peripheral VGA output

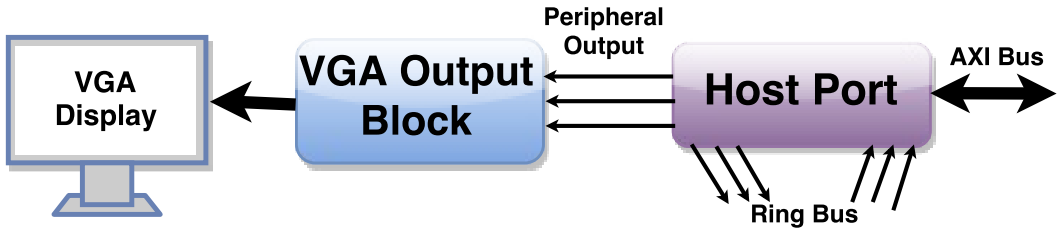

Figure 5.6: VGA output block attached to Host Port

To use the capacities of the Host Port's output peripheral interface, a VGA output block was connected to it, as represented in Figure 5.6. This IP block was mostly reused from another project implemented on another hardware platform. The inputs and the outputs had to be redesigned to fit in to this implementation, but the main logic was used as it is. The modifications that have been performed to the VGA block will be further presented in the following section.

5.2.1 VGA block Input and Output signals

Input signals

The VGA block has the following input signals: CLK, DIN and DSTR. With these signals it is possible to write 16 bits of data in DIN and signalise the valid data with DSTR.

The host port transmits 21 bits of data in 7 clock cycles. To convert this data in to the VGA block's input of 16 bits, a logic block is needed in between.

The peripheral bus coming from the host port stays in IDLE state when all the bits of the bus are 0, such that when one of the bits changes to a 1 the transmission starts and the data begins to flow. At this point, 7 clock cycles are needed to deliver the total 21 bits of data. For the VGA block only the 16 least significant bits are of interest. This data is saved in a 16 bit variable and sent to the VGA block, signalising it with DSTR for one clock cycle. The timing diagram of this block is shown in Figure 5.7.

The logic block who performs this conversion is fed with the same clock source as the VGA block and the host port.

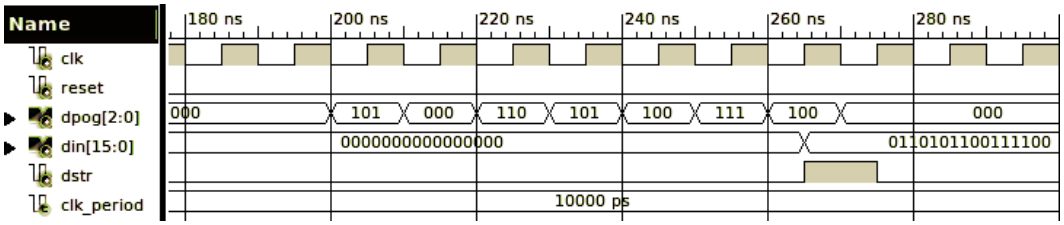

Figure 5.7: Timing diagram of the VGA input conversion block

Output signals

The output signals had to be adapted to the physical DAC present in the ZedBoard, as the original project had a DAC configuration that was 2 bits width for each colour. In the case of the ZedBoard the DAC is 4 bits width for each colour. Allowing a ampler variety of colours, nevertheless, for this application are so many colours not particularly significant, because the output will just be plain text.

The VGA block's output signals are the three colour signals, red, green and blue, and the vertical and horizontal synchronisation signals, where the synchronisation signals didn't need any adaptation, in contrary to the colour signals that were increased to 4 bits width. To maintain the logic that changes the background and foreground colours, a simple adaptation had to be carried out to the registers who store the display colours. These registers were also increased to 4 bits width and they were linked to the logic who handles the colour change. This logic was prepared for the 2 bits colour change, to be possible to change the colour with the same 2 bits, were used the registers's most significant bits.

5.2.2 VGA block initialisation

To produce a higher resolution on the display, the variant of 1024x768 was chosen.

In order to release the application from configuration features, some initialisation operations were integrated in the VGA circuit.

These operations generate automatic start up outputs, writing multiple characters in to the VGA block and configuring initialisation parameters.

In this initial start up, that is performed at system reset, the VGA circuit is configured with the 8 windows's positions and dimensions, such that any Application can immediately start writing characters in to the VGA block.

The Display is divided in 8 even squares. Each square providing its own output and character position management.

To signalise that the VGA block is ready, some characters are written on the Display at start up.

5.2.3 VGA block updates

With the exception of the mentioned blocks that were implemented to link this VGA block to the ZedBoard and to the Host Port, some minor updates were performed.

Some blocks presented a high hold and settle time in the timing reports, leading to possible failures. When investigating the failure in the problematic nets, the reset signal was found to be suspicious. As the logic blocks were analysed some wrongly asynchronous resets were discovered. They were changed to synchronous resets. And this solved some of the timing report problems that were found in this block.

Some other minor changes were performed to some of its logic blocks to improve reliability.

5.3 PiNets Ethernet Application for Desktop PC

In order to communicate, through the Ethernet interface, with the runtime system running on the Zynq, a simple PC application had to be created.

With this application it is possible to:

- Start a Pi-Nets Application on the Zynq.
- Download Pi-Nets Applications and save them to the SD card.
- Write data in to the DDR memory.
- To send and receive data from the Host Port.

As this project was developed under Ubuntu, this application was developed and tested considering this platform. But this Application, was thought to be executable under Linux and Windows, with minor code changes. To achieve this cross platform functionality, the Pcap library was used. This library provides functions for the Ethernet communication, and is executable under Linux. For the Windows Systems the WinPcap library should be used, this library has almost the same functions as the Pcap library.

How the mentioned features were implemented will be further explained.

5.3.1 User Interface

The User interface was kept simple, with a text based GUI, visible in Fig.5.8. The available options are displayed in the terminal, and the user is asked to input the number of the message to send. As the user types in the desired message number, the application prompts to the function who constructs the selected PiNets message. When this task is completed and the Ethernet Packet was sent, the application returns to the menu entry, displaying the options again in the Terminal. Once here, the application is ready to configure and send another message.

To verify that the interface is properly connected and active, message 0 can be sent, in this way, the Zynq is forced to respond with message 1.

The options listened with an (*) symbol, in Fig.5.8, are messages that are received from the Zynq.

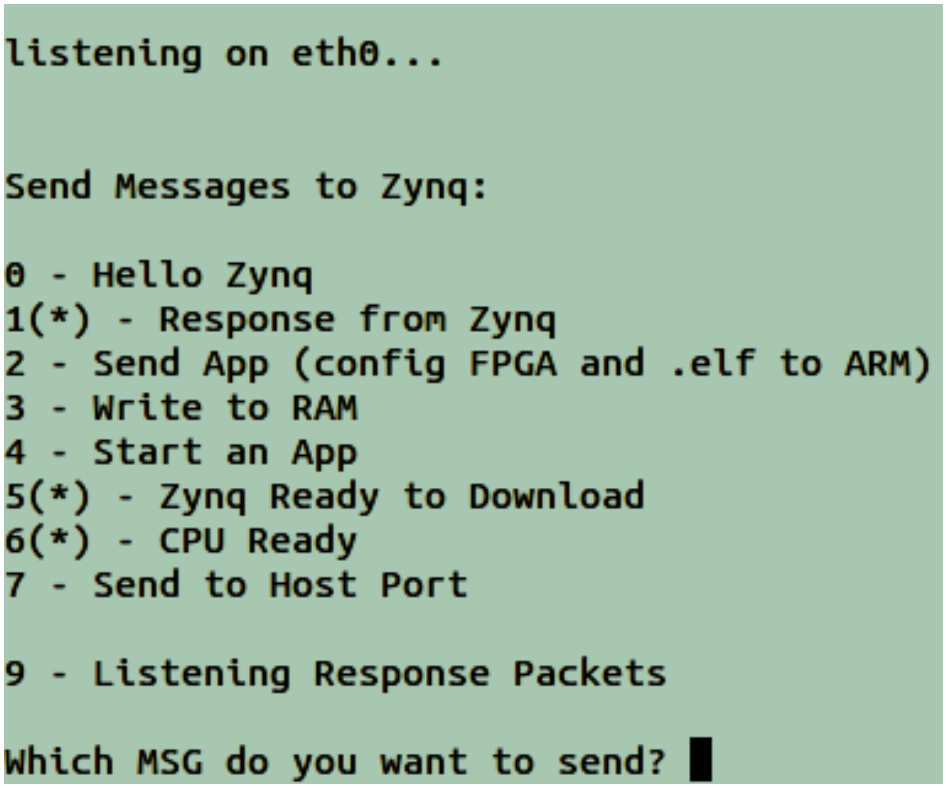

Figure 5.8: User Interface Menu

5.3.2 Starting a PiNets Applications on the Zynq

To start an application on the Zynq, message number 4 must be selected, by entering the number 4. To set up this message, the user will be asked to type the information of the Application to be executed.

The first required information is the Applications Name, maximal 30 characters to be used. As soon as the name has been introduced, the app Code of the Application will be asked, and finally the app Version will have to be entered.

The Ethernet massage is build based on the PiNets Ethernet communication Protocol. Where the raw Ethernet frame is constituted by an Ethernet Header, a PiNets header and the typed information.

0x4	app Version	0	app Code	0	app Name	Zynq MAC addresses

Table 5.3: PiNets Message 4 frame

When the Zynq receives the packet and processes its content, it will check if the instructed application is present in the SD card. If the application files are found on the SD card of the Zynq, it will automatically configure the PL and start running the selected PiNets Application in the PS.

If the application is not present in the SD card, the Zynq will send, over the Ethernet interface, a request to download the files of the ordered Application. This request is received in the form of a PiNets message 5, and contains the information of the Application that the Zynq requests to download.

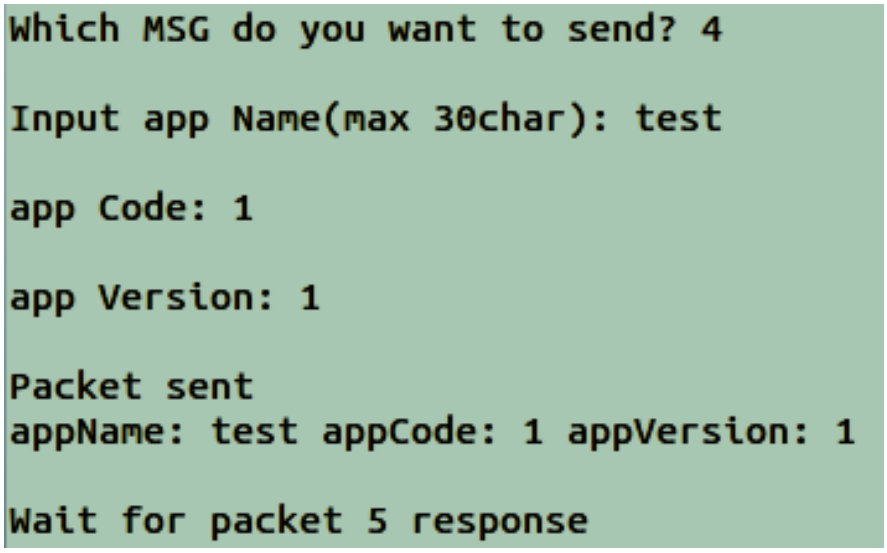

Figure 5.9: Starting an App on Zynq

5.3.3 Loading a PiNets Applications to the Zynq

To send application files to the Zynq, a message of type 5 must have been received, where the Zynq requests the application files. If this is the case, the application files can be loaded by selecting the message number 2.

The user will be asked if a Epiphany executable file is to be loaded. Further updates to support the Epiphany will be developed later, as optional feature.

The application is constituted out of three files that need to be sent to the Zynq, they are the FPGA configuration file, the ARM core0 .elf executable file and the ARM core1 .elf executable file.

To construct the PiNets message of type 2, the user will have to input the Name, Version and AppCode of the application files. The files must be stored in the same folder as the application that is running on the terminal. The names of the files will have to respect the following name structure:

- FPGA configuration files: app<AppCode>_v<Version>_<Name>.bin
- PiNets Program Code ARM0: app<AppCode>_v<Version>_<Name>.arm0.elf
- PiNets Program Code ARM1: app<AppCode>_v<Version>_<Name>.arm1.elf

If the user does not input a valid information of the application files or if one of the application files does not exist on the path, the Application will fail and no files will be sent. Even if one or both ARM cores don't need to process any meaningful code, a program code needs to be sent. An example is an infinite loop.

The First Ethernet packet that is sent contains information regarding the size of each file. For the size of each file are 4 Bytes reserved, the rest of the packet is filled with data. A data stream of Ethernet packets is then created, having each packet 1024 Bytes and consisting of the FPGA configuration file and the executable for both ARM cores. These packets do not cross the files data, if the data of a file has come to its end and the Ethernet packet does not have the 1024 bytes filled, then it will send the frame as it is. The files are sort from the Least significant Byte to the Most significant Byte.

After the files have been successfully sent, the application will immediately start on the Zynq.

```
Which MSG do you want to send? 2

Is the Epiphany file also to be sent? (Y/N)
n

The Pi-Nets application files must be under the same
path as where this application is executed!!!
The names must have the following formats:
- (app<AppCode>_v<Version>_<Name>.bin):

- (app<AppCode>_v<Version>_<Name>.arm0.elf):

- (app<AppCode>_v<Version>_<Name>.arm1.elf):

Input app Name(max 30char): test

app Code: 1

app Version: 1

 Files sent
```

Figure 5.10: Loading Application to Zynq

5.3.4 FPGA .bin File and PiNets Program Code Format

To generate an FPGA configuration file that can be handled by the runtime system, a conversion has to be taken care. The FPGA configuration file is generated in the .bit format, but the file will have to be converted to .bin format, aligned to 32 bit. When we generate a .bit file from an implementation, this .bit file will include a small header and the bitstream to program the FPGA. The header must be removed, and it can be configured when generating the .bit file or when converting the file to .bin. At last, the configuration file must be submitted to a byte swapping.

To convert the generated .bit file to .bin, using the Xilinx ISE tools, the following command is to be used:

- promgen -b -w -p bin -data_width 32 -u 0 <bitfile.bit> -o <binfile.bin>

Using the Xilinx Vivado tools, the following code line should be inputted in to the Tcl console:

- write_cfgmem -format bin -interface SMAPx32 -disablebitswap -size 128 -loadbit "up 0x0 <bitfile.bit>" -file <binfile.bin>

The piNets program code files must be compiled specifically to run under the ARM architecture. Some care must be taken when compiling the Elf file. The linker script that compiles the Elf file must link the program at the right start address. This address is defined by the piNets runtime system, as the PiNets application code start address in Table 2.3.

5.3.5 Writing in to DDR memory

To assist the PiNets Program running on the Zynq, it is possible to write in to multiple DDR memory addresses using only one Ethernet packet. To accomplish this, the PiNets message 3 must be selected.

By selecting this message type the user will be asked to input the number of the app Code from the PiNets Application, and the Data that will be written to memory will be associated to this app. After this step, the user has to type in the number of addresses plus the number of words that he wants to send. Before the user types the addresses and respective Words, the total number of each must be typed in. The minimum allowed is 2, such that one address and one Data Word is sent. Each address or Word is 4 bytes wide and the input has to be in hexadecimal.

Assuming the User inputs a 2, to input one address and one data Word. The next step will be to input the address. After the address has been typed, the Data Word follows it. At this point, the Ethernet Packet had gathered all the needed data and would be sent. The most significant bit of the address in this case is 0.

Frame structure:	0x3	N	0	App Code	Address 1	N Words or addresses

Example structure:

0x3	6	0	1	Address1 = 0x**8**0004120	Data1 = 0xFAFAFAFA	...
...	Address2 = 0x**8**0005678	Data2 = 0x10101010	...			
...	Address3 = 0x**0**00F9876	Data3.1 = 0x33331111	Data3.2 = 0x33332222			

Table 5.4: PiNets Message 3 frame structure and example

In order to send multiple data words and respective addresses, the input has to be bigger than 2. In this case the Address's most significant bit has to be 1, such that the runtime system's decoding logic can identify it as an address. The data word follows the address, after that another address and data word can be typed in, and so on. The last address that is typed, must have the most significant bit with 0, in order to be identified by the runtime system's logic as the last address. The following convention has to be respected when writing multiple address and data words:

- The address is first typed followed by the data Word
- If it is not the last address to be typed in then the most significant bit must be a 1
- The last address that is typed must have the most significant bit with a 0
- After the last address has been given, multiple data Words can be typed, they will be written to the last address with its respective increment

```
Which MSG do you want to send? 3

Input app Code: 1

Input number of Addresses + Words: 7

1-ADDR: 80004120

2-ADDR or WORD: FAFAFAFA

3-ADDR or WORD: 80005678

4-ADDR or WORD: 10101010

5-ADDR or WORD: 000F9876

6-ADDR or WORD: 33331111

7-ADDR or WORD: 33332222

Packet sent Sucessfull
```

Figure 5.11: Writing to DDR memory

An example of the structure of such message type is represented in Table5.4. This example was typed and sent using the application, as represented in Figure 5.11. The addresses and data that were written with this example are shown in Table 5.5.

Address	Data
4120	FAFAFAFA
5678	10101010
F9876	33331111
F987A	33332222

Table 5.5: Written DDR memory addresses and respective data

5.3.6 Host Port

If we want to send data to the Host Port the PiNets message number 7 has to be selected.

The User will have to input 3 bytes, representing the 21bits of data that will be sent to the Host Port. The three bytes contain 24 bits of data, where the most significant bit is in the first entered byte. The three last bits of the last entered byte are discarded.

The constructed Ethernet frame is displayed to the user before it is transmitted, where the user is questioned if he wants to wait for an answer from the Host Port, as represented in Fig. 5.12. If the User responds with an “n”, the Ethernet packet is sent and no answer will be expected.

If in the other case the user wants to receive a response from the Host Port then, a “y” is given as input and the program will wait for an answer from the Host Port. As the answer is received its content is displayed on the terminal.

```
Which MSG do you want to send? 7

Write 3bytes in HEX for the 21bit Host Port Data (MSB first)
FEFAFB

The following 21bit Data will be sent: fefafb

|     Zynq IP     |       My IP      | Typ |     Data
 00.0a.35.01.02.03  84.8f.69.c4.78.c5  6712  848f69c478c512b9b0a107fefafb
Do you want to listen to a Response?(Y/N) n


Message sent to Host Port
```

Figure 5.12: Send data to Host Port

5.3.7 Listening Mode

Such that its possible to visualise all the received packets from the Zynq, a listening mode was added to the Application.

In this mode, the application is waiting for new packets to arrive, and, whenever a new packet arrives from the Zynq, the content is displayed on the terminal.

The listening mode can't be active at the same time as when the application is sending other messages, blocking the normal execution by waiting for incoming packets. As there were no simple method to implement a interrupt triggered response to the incoming Ethernet packets, that would be compatible with both Operating Systems. To overcome this limitation it is possible to start the listening mode in parallel, by starting the application on a a second Terminal window. And so be able to execute two equal applications where one launches the normal application and the other is configured in listening mode.

```
Which MSG do you want to send? 4

Input app Name(max 30char): test

app Code: 1

app Version: 1

Packet sent
appName: test appCode: 1 appVersion: 1

Wait for packet 5 response
```

Figure 5.13: Listening mode

5.4 Memory controller

To enable writing and reading data to and from the DDR memory by the components configured on the PL, a memory controller was specified.

This controller connects from the PL directly to the DDR, without using the APU or the DMA. This configuration is achieved by using the high performance ports of the Zynq, who communicate with AXI4 protocol.

The interface to the PL's components has to be simple and as efficient as possible. Using the lowest possible input/output signals, without reducing the functionality.

This implementation is meant to be reused multiple times and configured with different parameters, such as Data or address widths and burst sizes. Multiple High Performance Ports could be configured with this implementation.

5.4.1 High Performance Ports

The most direct path from the PL to the DRAM are the Zynq's High Performance ports specially designed to link the PL with the DDR memory. These ports are linked to the PS, where a DDR controller is responsible to share the access to the DDR memory, between the PL, the L2 cache and the DMA. These ports have the fastest throughput in transactions with the DDR memory. In order to address these ports is used the AXI communication protocol. The data width of each port can be configured with 32 bits or 64 bits. In this implementation the 32 bits configuration was used, nevertheless the 64 bits configuration would also be configurable with a small change.

In between the specified block's AXI interface and the High Performance port, was configured an AXI interconnect from the Xilinx IP Core. This IP block is used as a bridge between the two interfaces, carrying the signals of the AXI interface, and handling the addressing of the attached blocks. This AXI Interconnect also translates the AXI4 protocol to the AXI3 protocol, as they have minor changes in the width of some signals. The DDR controller block that was implemented uses the AXI4 communication protocol, but the High Performance ports from the Zynq must be addressed with AXI3.

5.4.2 DDR controller interface

The DDR controller's interface was kept simple, containing just the necessary signals for a Write or Read operation. This block gives an abstraction layer to the AXI protocol, using a custom designed input interface. It is intended that applications use this custom signals to write or read from the DDR memory, providing this block, the logic to convert the signals to the AXI communication protocol. The designed memory controller block and its respective signals are represented in Fig. 5.14. The signals starting with M00 refer to the AXI protocol signals, all the other signals make part of the designed custom signals. This custom signals will now be described.

For the Write operation the following signals were created:

Write To signalise the start of a write operation.

WData Carries the data that is to be written.

Writestatus To demand new data input, when in burst mode.

The signals that are used for a Read operation are:

Read To signalise the start of a read operation.

RData Carries the read data.

dataReady To signalise the availability of new data.

The signals that are used in both operations are:

Addr This bus carries the memory address, in which the operation is performed.

Burst This signal indicates the amount of burst beats that are to be transmitted.

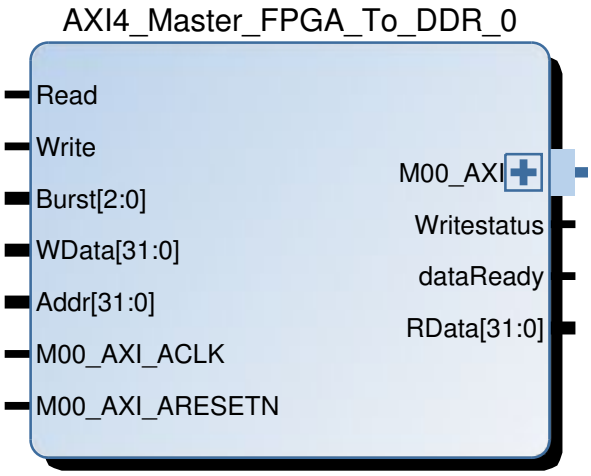

Figure 5.14: DDR controller block

5.4.3 Write Transaction

To initiate a write transaction, the signal *Write* has to be asserted, for one clock cycle, to 1. Before this signal is asserted to high, the signals *Addr* and *WData* must be updated with valid data.

The DDR memory address is indicated with the signal *Addr* and the signal *WData* is updated with the Data that is to be written to the respective address.

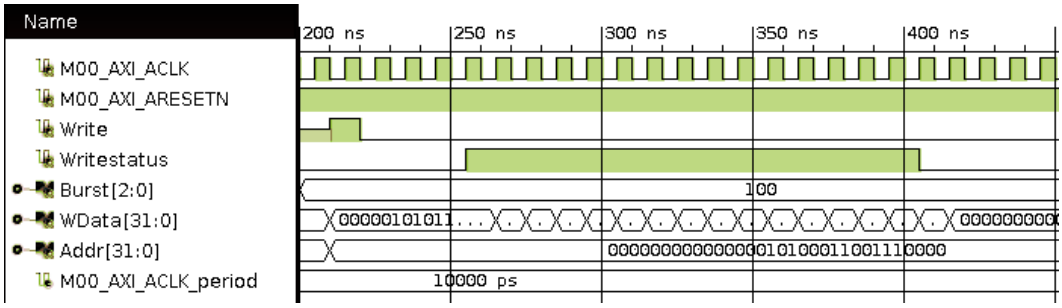

Figure 5.15: Timing diagram of the memory controller's write transaction

Conversion to AXI4 Master

The memory controller starts its operation in idle state, where the *Write* signal is 0. Before or at the same time that this signal is asserted to 1, the transaction signals, *Addr* and *WData,* have also to be asserted. The memory controller contains a state machine, who controls the start of the write transaction. As the *Write* signal is asserted the state machine changes its state and initiates the write transaction.

When the write transaction starts, all the counters of each data channel are first reset. After this reset, a new write operation can take place.

The AXI interface verifies the asserted address, considering the mapped memory range of the DDR memory, and, if the address is valid, a validation signal is asserted. After the validation of the Address signal is accepted, the data is asserted on the data channel, using the handshake signals to validate the transaction. To ensure that the data has been correctly

written, the response channel takes care of transmitting the write response, and identifying so possible errors.

As the write response is received, the state machine changes state to transaction complete.

In this last state the signal "ERROR" is updated. This signal is specially useful to identify problems, when debugging. In the last step of this same state, the state machine returns to the starting step, changing its state again to idle.

5.4.4 Read Transaction

Similar to the write transaction, to initiate a read transaction the signal *Read* has to be asserted, for one clock cycle, to 1. In case of a read, just the *Addr* signal has to be asserted with the address intended to be read. The data that is read from this address, will be asserted to the signal *RData*, signalising its arrival with the signal *dataReady* to high, for one clock cycle. The components that use the interface read the content of *the RData* signal when the *dataReady* signal is asserted high.

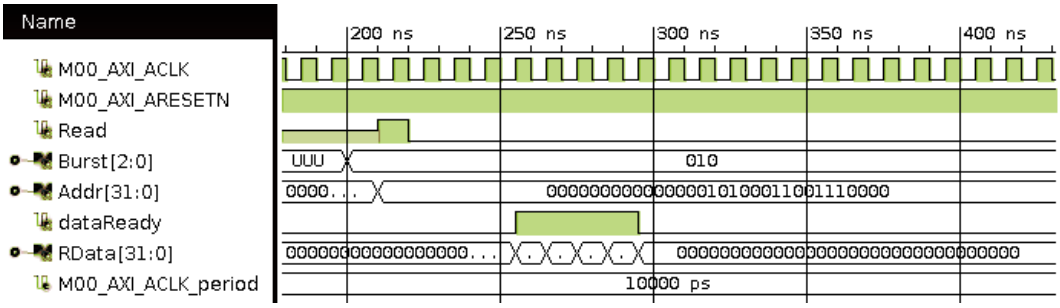

Figure 5.16: Timing diagram of the memory controller's read transaction

Read converted to AXI4 Master

When the *Read* signal is asserted to high, the state machine changes its state to initialise the read transaction.

The Address has to be asserted as the *Read* signal is asserted, to correctly process the handshake signals, as the address channel is validated by the change of state. When the Address is validated and the Data is made available, the controller forwards the data to the *RData* signal and sets the *dataReady* signal to high for one clock cycle. The data can then be read by the attached component.

By the end of the transaction, the state machine changes its state to complete, and processes the same logic as in the write transaction to return to idle.

5.4.5 Burst Mode

In order to use the HP port with its highest throughput performance, a burst feature was included in the implementation. Whit this burst option, it is possible to burst multiple Data Words successively, at each clock cycle, being it accomplished by just sending one address to the DDR controller.

The AXI protocol specifies different ways to increment the addresses automatically. In this implementation the increment configuration adds a value of 4 to the address, in each beat

of the burst. As the width of each data Word is 32 bits and the memory is byte addressable, the increment must be 4 at each beat.

The number of data Words that are written or read in the burst transaction, are set with the *Burst* signal. This *Burst* signal is three bits width, allowing to define 8 different burst sizes, represented in Table 5.6. The signal was defined to grow in size with the power of 2, to serve different needs of interaction with the DDR memory.

Burst signal	000	001	010	011	100	101	110	111
Burst size	1	2	4	8	16	32	64	128

Table 5.6: Burst signal in relation to the burst size

The *Burst* signal must be set at the moment were the write or read instruction is given, to initialise the burst configurations.

To integrate the burst feature in the Write and Read transactions, some adaptation had to be configured in order to handle the burst data transaction with the custom signals. The implemented functionality for both transaction types will be further clarified.

Burst Write

When a Write transaction is initiated in burst mode, being the *Burst* signal different than "000", the normal Write transaction configurations have to be set at the beginning, where the *Write*, *Addr* and *WData* signals have to be asserted.

To send the remaining data Words of the burst, the signal *Writestatus* was added to the custom signals. This signal is set to high at the moment that new data Words are expected, and it will remain high for the necessary clock cycles to perform the ordered burst. The attached component is supposed to assert the data Words, at each clock cycle, and for the remaining ordered burst size, at the moment the *Writestatus* signal is high, as it will remain high until the last data Word is awaited.

Burst Read

Similar as a normal Read transaction, on performing a Read burst the same initial signals have to be asserted, and also the *Burst* signal asserted with a value different than "000".

The same configuration as a normal read transaction applies when performing Read burst transaction, as the *dataReady* signal is asserted for one clock cycle when new data is available. In the case of a burst, the data Words will arrive sequentially, and also the signal *dataReady* will in this case be high for the clock cycles of the requested burst size.

5.5 Porting to Parallella Board

To continue the progress of this thesis, the development hardware used until now had to be changed. The ZedBoard was a good Development board, in the sense of its wide range of configurability and many different peripherals. The Parallella board on the contrary, is not that configurable and does not have so many peripherals. Nevertheless, to have access to the Epiphany co-processor, this board will be needed. The Epiphany chip has very attractive specifications, being able to deliver a high processing throughput with its 16 cores. One of the goals of the infrastructure is to integrate the Epiphany co-processor in the project. To

be able to use the components that were designed until now, the Runtime System that runs on the ZedBoard has to be Ported to the Parallella Board.

5.5.1 Boot process

The Parallella Board has two ways of booting the PS and consequently configuring the FPGA, using the QSPI flash or the JTAG connection.

As default, the QSPI is flashed at Production with an FSBL and the U-Boot application. The FSBL is specific to the Zynq and can be compiled with the available Xilinx Tools in SDK. The U-Boot application can perform multiple tasks, but its main task is to boot a Linux image, offering a very reliable way to do it. But when an .elf application is to be booted, case of the Runtime System, this is not possible with the U-boot Application.

The only remaining option to configure the Parallella board, not using a linux based OS, is to connect a JTAG cable, and so be able to debug the PS at real time and to configure the FPGA when ever necessary. With a JTAG cable it is also possible to flash the QSPI memoy and so be able to boot a different application than the U-Boot.

5.5.2 Providing the JTAG pins

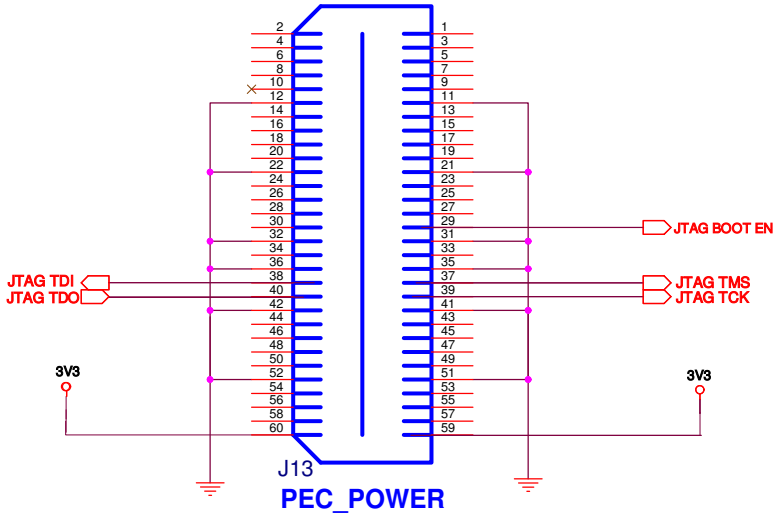

Figure 5.17: PEC-Power's JTAG pin location

The Parallella board, does not offer the JTAG pins in an accessible connector, like USB, being, instead, available in a GPIO connector. The Parallella has 4 GPIO connectors, and the one that transports the JTAG pins is the PEC-Power.

The board that was used in this Thesis had these connectors not populated. Considering that the needs of a JTAG connection was to high, the pins were manually soldered to give access to a JTAG capable cable connector. The JTAG pins were provided according to Figure 5.17, where the schematic of the soldered pins is shown. The composition of the JTAG signals are TDI, TDO, TCK, TMS, GND and power. There is also a pin available to configure a JTAG boot. When this pin is left unconnected proceeds the QSPI boot.

5.6 Epiphany integration and control

To integrate the Epiphany co-processor in the design, a proper hardware platform has to be created. The Epiphany chip is placed externally to the Zynq and linked through the East interface of the chip, but wired to the PL's IOs. To interface and communicate to the Epiphany chip, the eLink Protocol is used.

The hardware that implements the eLink Protocol is available in the Parallella Github repositories, and even that some adaptions had to be taken care of, most of the logic is ready to be used. This hardware project also interfaces other peripherals present on the Parallella board, like HDMI and I2C, but for this implementation just the Epiphany control hardware is of interest.

Once the hardware is integrated in the project, one can create a new BSP, to support the Software that is going to be executed on the ARM. This BSP contains a definition of the addressable path to the Epiphany component. Also the PiNets system has to be reconfigured with this new BSP, and any specific ZedBoard functions have to be adapted.

Once the development environment is prepared, the Epiphany can be viewed more specifically. Part of the integration of this component is to initiate the execution of instruction code in its individual cores. Once the chip is processing data, one would like to read and write from its memory registers, for control and for data acquisition.

The most important functions in the integration of the Epiphany chip are the loading of instruction code and the reading and writing of individual and multiple memory registers, in which the loading of instruction code will be accomplished by the PiNets Runtime System, as it initialises a normal PiNets application. The read and write functions are available for software and hardware access. Other and more specific functions could also be incorporated in both systems, depending on the needs.

The Github repository also contains the Software sources that enable the communication with the Epiphany chip, among many others. These sources, however, were designed to be executed on a Linux machine. Having in its composition the need for many Linux supported libraries. Some of these functions can be adapted to run under the PiNets runtime system, and so be used on a PiNets applications that makes use of the Epiphany building block.

The steps of the integration of the Epiphany co-processor, will be explained in the next subsections, in more detail.

5.6.1 Integration of eLink hardware and interface Epiphany

The integration of the Epiphany block in the PL is based on the Github project found on [29]. This project's sources are the official, open source licence, Parallella hardware platform configured on the PL, which links the hardware peripherals present on the Parallella board to the PS ports. This project was used as a start, and, on top of it, some additional blocks were included. Some blocks had to be adapted and minor changes were performed to configure the right chip. The main configuration specific to the Parallella board was already performed by this project. The blocks, that make part of the infrastructure, were integrated in to the design. In order to incorporate the Epiphany access port on the PL, some blocks needed to be adapted, and so be able to access the Epiphany memory structure within PL logic.

This project implements the eLink Protocol, linking, this way, the PS to the Epiphany chip. The Epiphany IO signals are based on RX and TX, both are 12bit width and they carry the mesh data along the interface. The eLink block has three links to the PS, each of them through different ports, with distinct AXI protocols as communication interface. The three interfaces, that are represented in Figure 5.18, are:

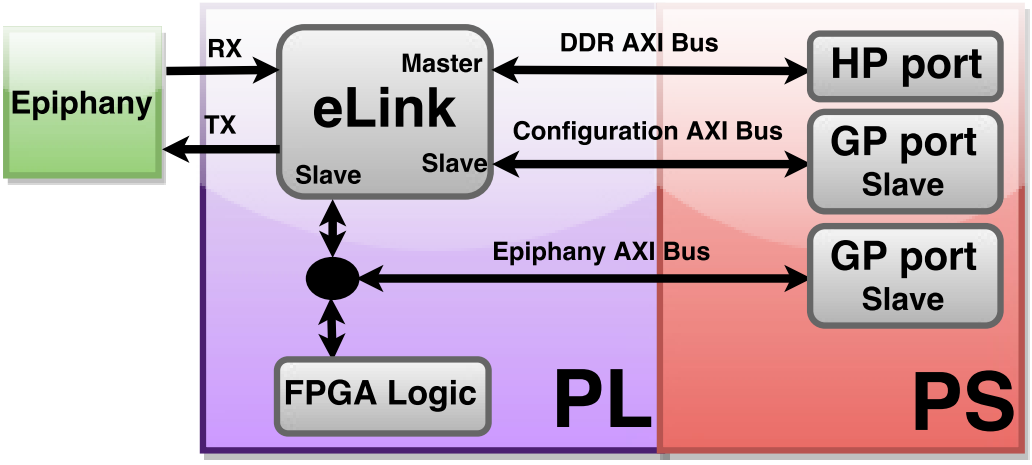

Figure 5.18: eLink hardware connections

- DDR AXI Bus: Interface to link the eLink hardware directly to the DRAM through a High Performance port, using AXI4 in Master configuration.
- Epiphany AXI Bus: An interface to link the Epiphany block, configured in slave mode and accessed with AXI4. Ranging its address space between 0x80000000 and 0xBFFFFFFF.
- Configuration AXI Bus: This interface is also configured as slave, but with AXI4-Lite. This interface enables the configuration of some registers for the initialisation of the Epiphany chip. The registers are located on the PL, and they are mapped with a memory range of 8K and with a offset address of 0x70000000.

The interface that links to the Epiphany AXI bus had to be adapted, in order to give the same access to PL logic, as its default design was configured to be accessed only by the PS. An AXI interconnect was used in between to solve this problem, where both PS and PL are masters and the Epiphany chip is treated as a slave acting like a memory block. Both have access to the entire memory range of the Epiphany chip.

To write to the Epiphany registers, from the PL, one can use the already designed memory controller block and so write or read in the desired registers.

After the proper hardware configuration was achieved, the project was implemented and its hardware platform exported to the SDK platform. A regeneration of the BSP was necessary to include the addresses of the new attached peripherals in the Pi-Nets runtime system.

5.6.2 Initialisation and start of the Epiphany co-processor

With the new BSP configured, the Pi-Nets runtime system's source files were added to the project. All its content was linked to the variable names contained in the BSP, such that almost no changes were needed.

To link the Epiphany as a processing unit in the runtime system, configurations have to be performed, by initialising the chip, loading instruction code to its memory and start its execution. These functions are integrated in the runtime system, in the flow of the configuration of the ARM cores, being the last task before the PiNets application's handoff.

The initialisation of the chip is based on sources provided in the Epiphany SDK, the e-hal library, were a routine to reset the Epiphany platform is executed. However, the routines of

the e-hal library were not possible to be compiled in the Pi-Nets runtime system environment, as they were designed to be used under a Linux OS. The routine had to be rewritten to take the specific Linux OS parts out. By resetting the Epiphany platform, the north, south and west link are set to be disabled.

Once the Epiphany is restarted, the instruction code can be loaded in to the individual Epiphany core's memory. This is performed by using the function that configures the ARM cores's Elf files. Some small adaptations specific to the Epiphany had to be carried out, as there is needed some extra input to identify the Epiphany core's address. To refer to a specific core on the Epiphany, the line and column number must be indicated, in order to translate it to the corresponding core address. In Table 5.7 are the translations between core address and its column and line numbers represented.

	Column			
Line	0	1	2	3
0	0x80800000	0x84800000	0x88800000	0x8C800000
1	0x80900000	0x84900000	0x88900000	0x8C900000
2	0x80A00000	0x84A00000	0x88A00000	0x8CA00000
3	0x80B00000	0x84B00000	0x88B00000	0x8CB00000

Table 5.7: Epiphany Cores address table

This implementation loads the same Elf file to all Epiphany cores, but it would be easily adapted to load distinct Elf files to each core, if they would be loaded from the SD card. The file manager for this propose was not created. This decision was taken because the download, through the Ethernet interface, of one extra file was easily adaptable on the created tools.

The PC application was integrated with an option to load an Epiphany executable file, as represented in Figure 5.19. The name of the file has to be respected or the application will fail. If the 16 Epiphany cores had to be loaded with different applications files and loaded using the Ethernet interface, a tool had to be designed to perform such task.

To start the execution of the loaded instruction code, on the Epiphany, another function was adapted from the Epiphany SDK. This function writes in to a specific register of a core, the *ILATST* register, ordering that the core synchronises its function, by initiating code execution. The default start position is the first address of its local memory. If the code is to be initiated in another memory position, a jump instruction can be written in the first memory address to point to the code start address. This start function can be performed for an individual core, or for all cores together, but the same register must be individually written on each core. In this implementation all the cores are initialised right before the handoff to the Pi-Nets Application.

5.6.3 Epiphany routines

Some routines, from the Epiphany SDK, were essential for the integration of the Epiphany chip and to control its execution. These routines, as already mentioned, were designed for a Linux OS, having to be adapted to our runtime system environment. These routines can also be useful to communicate with the Epiphany cores, when designing a application that is to be executed on the ARM. The most important functions, that served as the fundamental integration of the chip, are the Epiphany reset and Epiphany start, both functions write in to some specific Epiphany register. In the case of the reset, also the configuration registers are written.

```
Which MSG do you want to send? 2

Is the Epiphany file also to be sent? (Y/N)
y

The Pi-Nets application files must be under the same
path as where this application is executed!!!
The names must have the following formats:
- (app<AppCode>_v<Version>_<Name>.bin):

- (app<AppCode>_v<Version>_<Name>.arm0.elf):

- (app<AppCode>_v<Version>_<Name>.arm1.elf):

- (app<AppCode>_v<Version>_<Name>.e.elf):

Input app Name(max 30char): test

app Code: 1

app Version: 1

 Files sent
```

Figure 5.19: Loading Application to Zynq with Epiphany support

The basic functions that were adapted and grant access to the entire Epiphany memory are the following:

- ee_read_buf(Row, Column, Address, Buffer, Size): Reads a memory region on the Epiphany's indicated core.
- ee_write_buf(Row, Column, Address, Buffer, Size): Writes a memory region on the Epiphany's indicated core.
- ee_read_reg(Row, Column, Address): Reads a memory register on the Epiphany's indicated core and returns the read data.
- ee_write_reg(Row, Column, Address, Data): Writes a memory register on the Epiphany's indicated core.
- ee_read_esys(Address): Reads a register on the eLink configuration hardware and returns the read data.
- ee_write_esys(Address, Data): Writes a register on the eLink configuration hardware.
- e_reset_system(void): Resets the entire Epiphany chip, by writing some configuration registers. Calls the functions disable_nsw_elinks() and enable_clock_gating().
  - disable_nsw_elinks(): Disables the north, south and west mesh links.
  - enable_clock_gating(): Enables the clock gating.
- e_start(Row, Column): Starts the indicated Epiphany core.
- e_start_all(): Starts all Epiphany cores.

The constructed routines can be used to access the Epiphany from the PS in a Pi-Nets Application environment.

5.7 Generated System

Putting all the components together can lead to some not expected problems, even if the individual blocks worked out well. A cause of this problem could be the clock frequency, if multiple clock sources are configured in the design. In this implementation the same clock frequency was used for all the FPGA blocks and it was 100MHz. This clock source is provided from the PS.

The configured inputs and outputs of the TOP level project must be mapped to the actual hardware. This is performed by adding a constraints file to the project prior of synthesis and implementation. This constraints file was provided by the board manufacturer.

The described design can not exceed the available hardware resources. In this implementation the consumed resources were very few, as the FPGA is to be filled with other logical components that will process data and make use of the components of the infrastructure.

5.7.1 Updates to the Pi-Nets Runtime System

As the functions of the Pi-Nets runtime system were verified, some malfunction was found. The Pi-Nets application that was loaded in to memory was not starting its execution. After further analyses and debugging, it was found out that the application was not being loaded to memory correctly.

By using an Elf file, as the application file, one has to consider that this file contains linking information in its header. The header contains memory addresses that indicate were the code has to be loaded in to memory.

The Elf file was being loaded in to memory without acknowledgement of such header. To overcome such problem an Elf loader was attached to the runtime system. This Elf loader would read the content of the Elf file header and load the instruction code to the right memory addresses. The source code of this Elf loader was adapted from [31].

Chapter 6

Testing and Results

In this chapter will be explained how the components of the infrastructure were tested and the obtained results will be discussed.

6.1 Testing the Infrastructure

To test the generated system, the individual components that make up the infrastructure will be put under proof.

The tests were designed to cover as much components as possible, in a single test environment. Nevertheless, this was not always possible as some components do not have any direct interaction between them. This was the case of the memory controller, which had its own test environment. The other components were tested using the Pi-Nets runtime system and the PC application as base, testing the entire interactions, from the boot until the start of a Pi-Nets Application.

Additional software features were added to the Pi-Nets runtime system, case of the Host Port and the Epiphany that have part of its functions integrated in the runtime system, being a crucial part of the test. The PC application is used to control the start of an application on the runtime system, among others. The tests that were performed will now be explained in more detail.

6.1.1 Host Port and peripherals test

To test the Host Port feature, combined with the VGA peripheral, the PC application is used and the message number 7 is selected. This message sends the inputted data to the Host port, testing so the entire interaction of the interface.

The Host Port on the PL distributes the received data through the ring bus or through the peripheral output. The ring bus is connected in loop, as no component's source code was available to test real interaction. The Host Port block that was reused in this implementation is a fully functional and tested block. The design of such test component overlaps the scope of this Thesis, so they were substituted by a feedback logic such that the same data that was sent would also return. The VGA output logic was attached to the Host Port.

When the Host Port on the PL receives the data, it is sent back by the loop, and intercepted by the PC application on the Ethernet interface.

When the Data is sent with the format on Table 6.1, the Host Port will dispatch the data to the attached VGA peripheral, and we can so, by inputting data through the PC application, send it along the way until it reaches the external VGA screen.

Peripheral code	not used	VGA input	Total
0x7	1bit	16 bit	21 bits

Table 6.1: Host Port input data for VGA output

This test is performed without any problem at the stage were the Pi-Nets application did not yet started. By initiating a Pi-Nets application the configurations of the interrupts are lost, as the handoff pushes the code execution to a different application, this application, who load a different interrupt vector table, located in a different memory space. To maintain the necessary configurations of the Host Port and other services offered by the runtime system, the best known option would be to include the initialisation of some of the main Runtime System's functions on the Pi-Nets application that is to be executed.

6.1.2 Memory controller test

A test environment was prepared on the PL to verify the function of the memory controller. To create such environment an AXI protocol capable test block was attached to one of the GP ports, to be accessed by the PS. This block simulates the behaviour of a component on the PL and is linked to the memory controller's custom ports. The test block is this way controlled by the PS, handling consequently, the memory controller. The diagram of such structure is represented in Figure 6.1.

Some C code was written to execute, on the ARM, simple write and read instructions to interface the test block, that would interact with the memory controller and so write to and read from the DRAM. Afterwards, using the APU, it verifies the content of the written addresses directly in the DRAM.

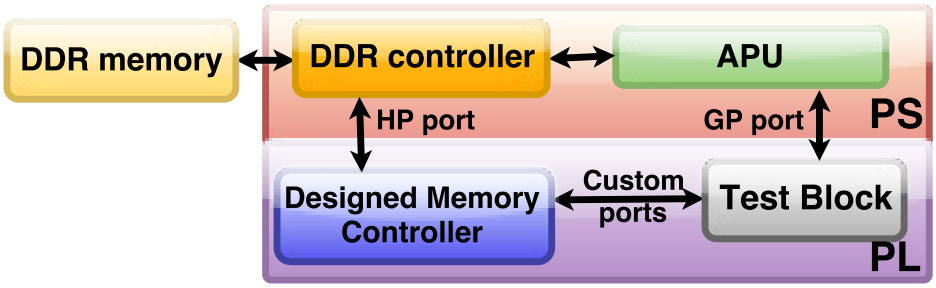

Figure 6.1: Diagram of the memory controller test

The test proved that the memory controller is working as expected, as the written memory addresses were afterwards read and the same data that was written was encountered.

6.1.3 Epiphany start and access test

The Epiphany was firstly tested isolated by using the modified Epiphany access routines, as resetting the Epiphany, loading the instruction code to its memory and starting its execution. The loaded code was based on some simple instructions represented in Listing 6.1, adapted from [32]. The first instruction is on the Interrupt Vector Table (IVT), that will jump to the first instruction code. Then, the value 0x80 will be loaded to the register r0, representing a specific local address, and in the register r1 is stored a 0. The next instructions will repeat itself over and over again. To the value present on register r1 is added 1 and its value is

```
;IVT:
0: b 0x40           ;0xE020

;Instruction code
64: mov r0,0x80     ;0x0310
65: mov r1,0        ;0x0320
Label:
66: add r1,r1,1     ;0x9324
67: str r1,[r0]     ;0x5420
68: b Label         ;0xE0FE
```

Listing 6.1: Epiphany test instruction code

stored in the previously loaded address indicated by r0. Finally, the instructions start over by forcing a jump instruction.

The 0x80 memory address is periodically read by the ARM, to verify its content and to see if the value increments. Once the required functionality was granted, in which the start, execution and control of the Epiphany chip was working, the integration was proceeded to the Pi-Nets runtime system.

6.1.4 Pi-Nets Application execution test with Epiphany

The Epiphany integration on the Pi-Nets runtime system was carried out to be able to load an application on the Epiphany chip, similar to the procedure of the ARM cores. Being the Epiphany, in this case, just another unit that has to be configured. To test the Epiphany in the Pi-Nets runtime system environment and to simulate the execution of a real Pi-Nets application, a test application was created. The test involves an application that would communicate between all cores, ARM's and Epiphany's.

This application is split in 4 files, FPGA configuration file from the created hardware project, Epiphany application file to be configured on all cores and ARM applications files for core0 and core1. The design of this test application will be further discussed.

The Epiphany cores were loaded with an application that would increment a specific address continually, such that this address could be read by one of the ARM cores. The epiphany code was compiled with the Epiphany SDK and an Elf file was generated. The application defines the core's local address 0x2000 as the address that is incremented.

The ARM core0 was configured as host, and it executes an application that reads periodically the values of the addresses that are incrementing on all the Epiphany cores. The read values are sent to the UART interface to be displayed. This core also communicates periodically with the ARM core1, as it sets a flag on an address in the OCM which will wake up the ARM core1.

The ARM core1 is checking the OCM address until the value is updated by the ARM core0 and afterwards sends some output to the UART interface.

This application executes on all cores correctly and outputs the expected UART output, but at some point, the application crashes and fails to deliver UART output. A reason for this failure might be the fact that the compilation of an Elf application using the Xilinx tools incorporates some initialisation and configuration functions that overlap with the initial Pi-Nets Application.

Because the deadline of this thesis was reached, and there was no more time to pursuit this error, the test was left as it is.

6.2 Obtained results

By using the Pi-Nets runtime system as main operating system many resources were spared, ideal to take the most of the system as it is a very low resources solution. Its functions bring a good and sufficient handling of the Zynq and control of the other peripherals, but there are also some limitations not having a full OS support. Some components, like the Epiphany, are not well supported using such a runtime system, making it more difficult to work with it. In this implementation the Epiphany was studied in more depth to cover the essential parts and to be able to integrate it with the runtime system. Also some other functions that could be reused from the open repository, would, in some cases, need some adaptations to work with the runtime system. Nevertheless, it was tested that the integrated Epiphany chip has all its cores operational and executing instructions.

Due to lack of time some errors have still to be corrected and further testing with more complete Pi-Nets applications should be carried on, as the test application, that was designed to test the Epiphany, does not entirely behave like it is supposed to. There might be some overlap in the initial configurations when the handoff of the Pi-Nets application occurs, so, the initial configurations should be taken care by the application that is initiated.

The individual components of the infrastructure were tested and seem to be working. Further individual tests could be implemented, but some issues could derive by joining components together. Testing the entire system is so of more interest.

6.2.1 Resources

Considering that the final FPGA projects are consisted of different blocks, represented in Appendix A and B, the consumed resources of the projects on both hardware platforms are the following:

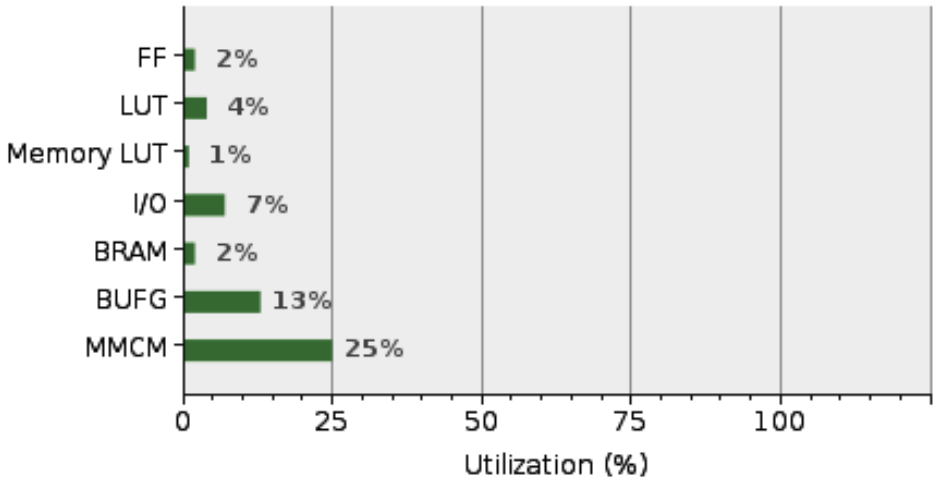

Figure 6.2: Project's consumed resources on the ZedBoard

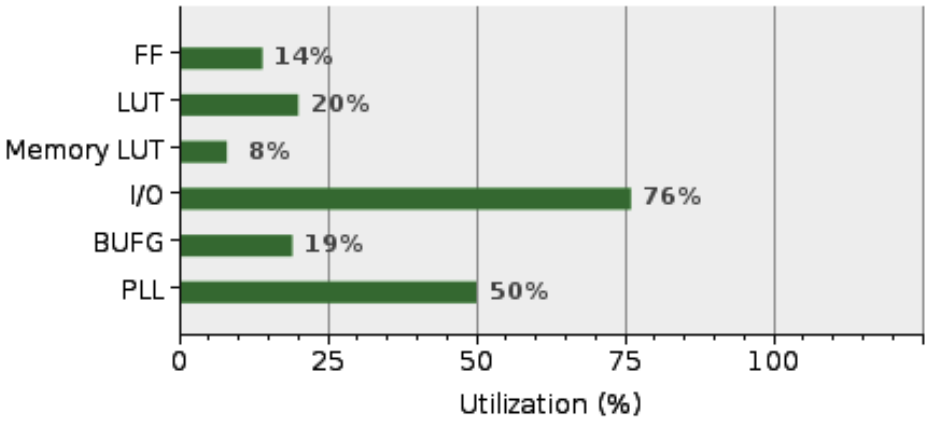

Figure 6.3: Project's consumed resources on the Parallella board

Chapter 7

Conclusions and Future Work

In this Chapter will be a conclusion of the global work that was developed, as well as proposals for improvement and suggestions for future work will be given.

7.1 Conclusions

To efficiently use the resources provided by the Zynq APSoC, some knowledge has to be presented or gained by the designer, to understand the individual areas of studies, in which such a system is enclosed. The designer is demanded to have knowledge and experience in digital design, FPGA architectures, FPGA design flow, PL-PS interfaces and also software development. After the sufficient knowledge from all areas was learned, the designer can focus on the problem and find solutions based on the available options and resources.

The main focus of this thesis was to find solutions to implement the components that make up the operating infrastructure, by reusing as many projects as possible. By reusing code developed under other projects one can save much time in the development phase and build so complexer systems, that integrate multiple interfaces and peripherals.

All the components of the infrastructure were verified and tested, and from a logical point of view, its main functions were working correctly, despite some errors could still exist. More testing should be performed, some ideas of a testing application is described in the next section.

The designed infrastructure can be used as it is, with the Zynq APSoC, or its functions can also be adapted to other platforms and other needs. The source code for both hardware implementations is provided in an attached media.

At last, it can be concluded that the main objectives proposed in this project were achieved successfully, as all the proposed blocks were implemented and since it was possible to demonstrate the interaction of the multiple components of the operating infrastructure.

7.2 Future Work

To further test the functions of the runtime system and to explore the created infrastructure, different Pi-Nets applications could be created. An essential application would be to test and verify the availability of all interfaces present in the system, as processing units, external peripherals and components configured on the PL, where some calculations could be performed on the processing units, and other components could forward the results to the external peripherals. This could be ordered by the host PC with specific input data.

The communication between the available processing nodes can be configured in a reserved memory space. The individual nodes could also transmit data to the external peripherals to display the progress of the application.

The Runtime system can at this point just execute one Pi-Nets application at a time, to initiate another application the system has first to be powered off. This functionality could be improved by integrating a general reset, where no power off would be required, being able to load an application almost immediately and at any time of execution.

The developed system, with exception to the Epiphany, can be ported to the Altera SoCs on the ER-4. This Hardware board has some differences considering it has a chip produced by a different manufacturer, but most of the specifications are similar to the Zynq chips. The logic of the project should reproduce the same results on both platforms, with possible slight differences. The biggest differences would be the tools that need to be used to configure the project on the board. Both fabricators have their own tools to handle the chip configuration. Once the configurations are settled, the project can be viewed as components described by VHDL and C code that can be integrated in to a similar platform.

The implemented Video output uses the VGA interface, that could be upgraded to the HDMI interface, as the HDMI interface also integrates support for audio channels.

Bibliography

[1] ARM, *AMBA AXI Protocol v1.0 Specification*. http://nineways.co.uk/AMBAaxi_fullspecification.pdf.

[2] Xilinx, *Zynq-7000 All Programmable SoC Diagram*. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[3] ZedBoard.org, *ZedBoard*. http://zedboard.org/product/zedboard.

[4] Parallella.org, *The Parallella Board*. https://www.parallella.org/board/.

[5] F. Mayer-Lindenberg, *Parallelrechner ER-4*, 2015. https://www.tuhh.de/ict/forschung/parallelrechner-er-4.html.

[6] Adapteva, *Epiphany Architecture Reference*, 2014. http://www.adapteva.com/docs/epiphany_arch_ref.pdf.

[7] V. Patki, *FPGA Basics and Architecture*, 2012. https://vedikapatki.wordpress.com/2012/05/01/fpga-basics-and-architecture/.

[8] W. Brandt, *FPGAs*, 2012. https://www.tuhh.de/t3resources/ict/dateien/Lehre/Hardware-Praktikum/FPGAs.pdf.

[9] H. M. Umer Farooq, Zied Marrakchi, *Tree-based Heterogeneous FPGA Architectures*, Springer (Ed.), Springer, 2012, Chapter 2.

[10] Corelis, *What is JTAG?* http://www.corelis.com/education/What-Is-JTAG.htm.

[11] Xilinx, *AXI Reference Guide*, 2012. http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf.

[12] Xilinx, *AXI Interconnect v2.1*, 2015. http://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf.

[13] Xilinx, *Zynq-7000 All Programmable SoC Overview*, 2015. http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[14] Xilinx, *Zynq-7000 All Programmable SoC - Technical Reference Manual*, 2015. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf.

[15] Xilinx, *Zynq-7000 All Programmable SoC Software Developers Guide*, 2015. http://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf.

[16] Avnet, *ZedBoard Hardware User Guide*, 2014. http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf.

[17] Parallella.org, *Parallella-1.x Reference Manual*, 2014. http://www.parallella.org/docs/parallella_manual.pdf.

[18] Y. Gevorkov, *Dokumentation des Pi-Nets Laufzeitsystems fuer das Zedboard.*

[19] J. F. Wakerly, *Digital design principles and practices*, NJ Pearson Prentice Hall, 2007.

[20] P. Mehta, *VHDL Syntax Reference*, 2003. http://webdocs.cs.ualberta.ca/~amaral/courses/329/labs/VHDL_Reference.html.

[21] Xilinx, *Vivado Design Suite User Guide*, 2014. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug910-vivado-getting-started.pdf.

[22] Xilinx, *Xilinx Software Development Kit (SDK) User Guide*, 2015. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/SDK_Doc/index.html.

[23] Adapteva, *Epiphany SDK Reference*, 2013. http://adapteva.com/docs/epiphany_sdk_ref.pdf.

[24] A. Neundorf, *Welcome to CuteCom*, 2009. http://cutecom.sourceforge.net/.

[25] Xilinx, *Linux*. http://www.wiki.xilinx.com/Linux.

[26] S. Rodrigues, *VGA text output with the Spartan-6 FPGA*, Project work, Technische Universitaet Hamburg-Harburg, 2015.

[27] S. M. Van Jacobson, Craig Leres, *PCAP*, 2014. http://www.tcpdump.org/manpages/pcap.3pcap.html.

[28] winpcap.org, *WinPcap user's manual*, 2007. https://www.winpcap.org/docs/docs_40_2/html/group__wpcap.html.

[29] A. Olofsson, *parallella 7020 headless Vivado Project.* https://github.com/parallella/parallella-hw/blob/master/fpga/vivado/releases/parallella_7020_headless.xpr.zip.

[30] M. R. Nigam, *AXI Interconnect Between Four Master and Four Slave Interfaces*, 2014. http://www.ijergs.org/files/documents/AXI-54.pdf.

[31] D. Sven Peter, svpe, *Front SD ELF Loader.* http://wiibrew.org/wiki/Front_SD_ELF_Loader#ChangeLog.

[32] rowan194, *Is there anything like e.load() which loads from memory?*, 2015. https://parallella.org/forums/viewtopic.php?f=23&t=2113&start=0.

Appendix A

ZedBoard's hardware block design

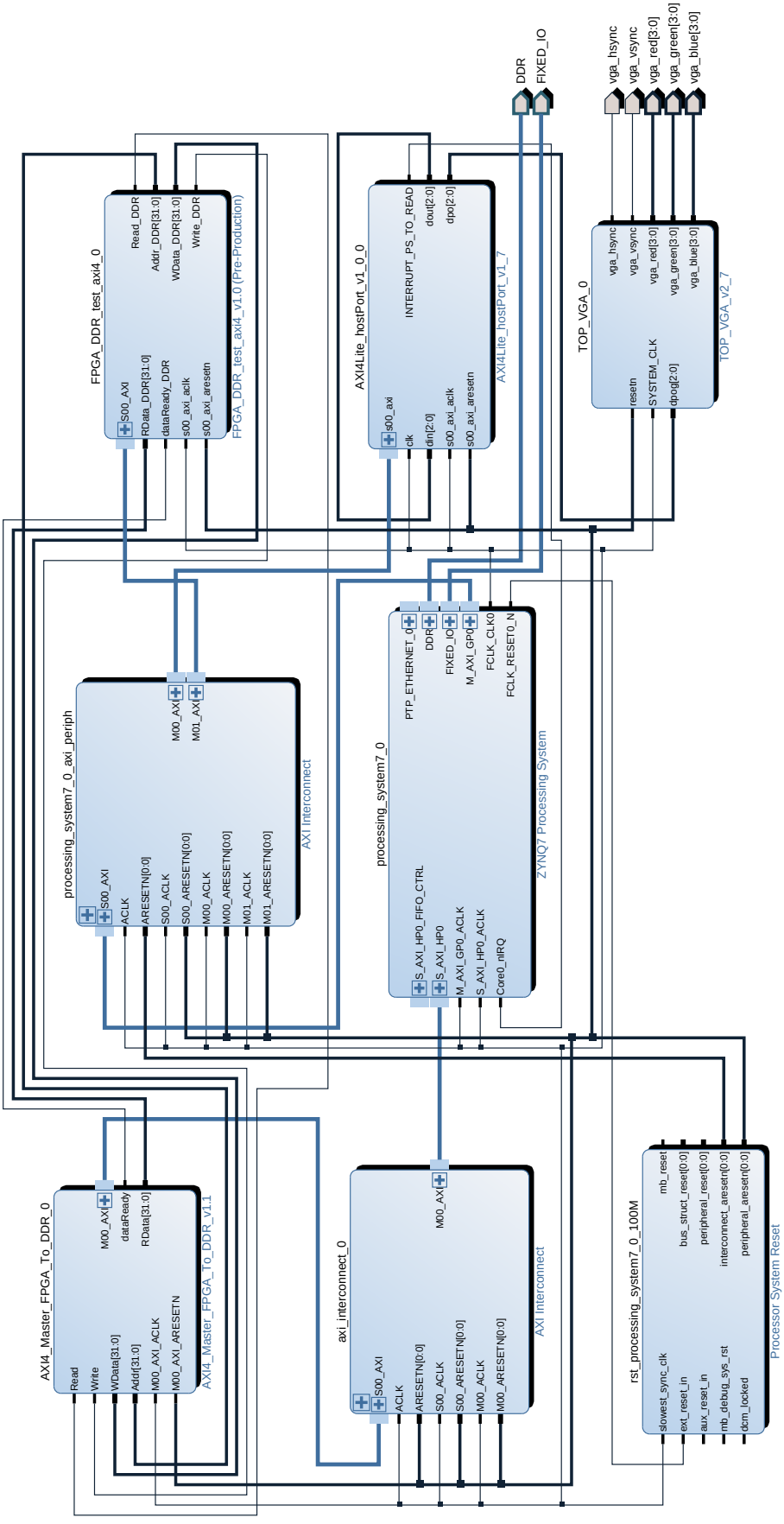

Figure A.1: ZedBoard's hardware block design

Appendix B

Parallella's hardware block design

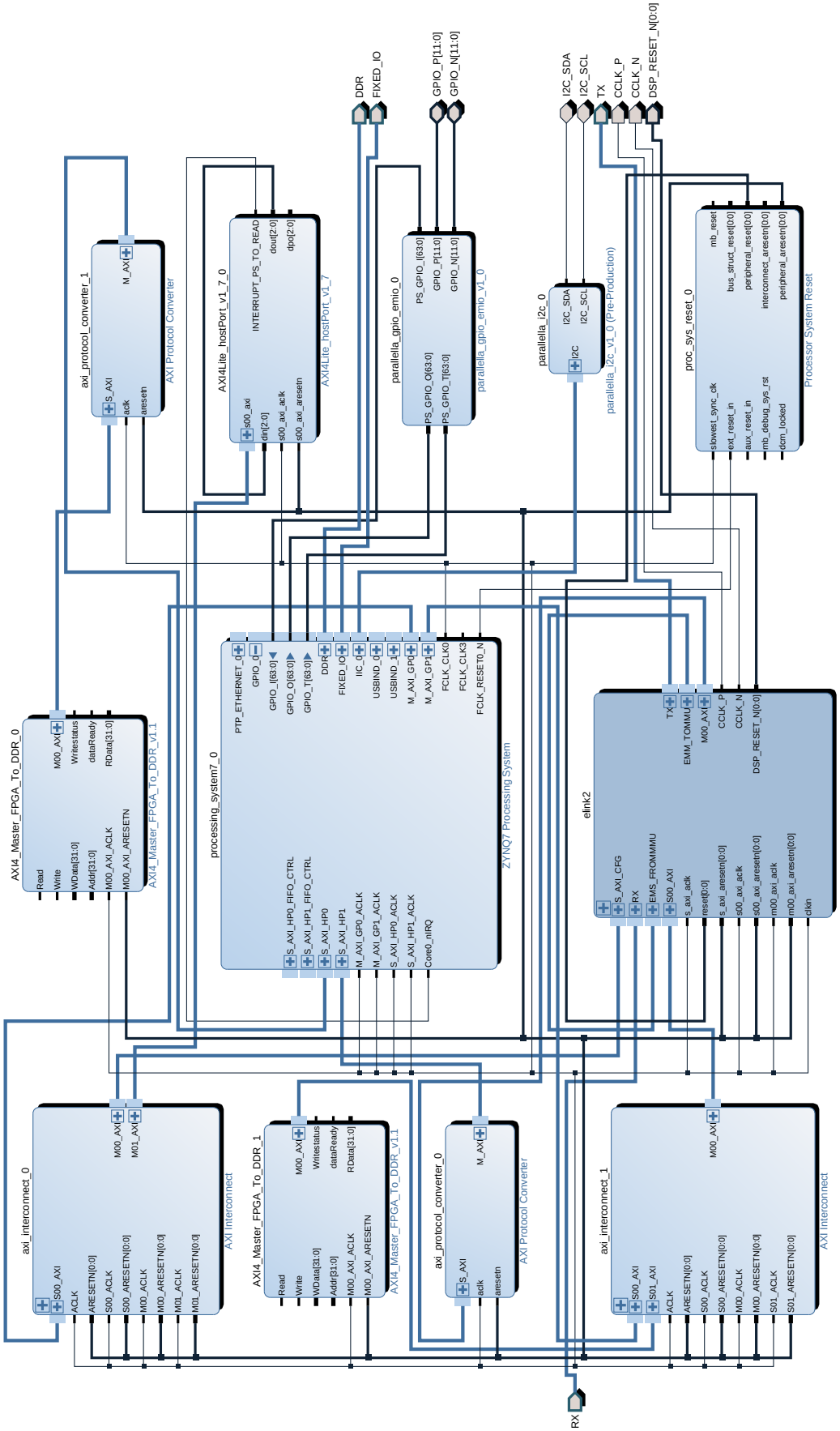

Figure B.1: Parallella's hardware block design